

PROJETO ACCOUNTOWNER – FRONTEND

Prof. Ms. José Antonio Gallo Junior

Componentes do Angular e Preparação do Projeto

A preparação de um projeto Angular pode variar de projeto para projeto e de versão para versão, este tutorial é dedicado a esse tópico.

Criar a parte do servidor (parte da API Web do .NET Core) é apenas metade do trabalho que queremos realizar. A partir deste ponto, vamos mergulhar no lado do cliente do aplicativo para consumir a Web API e mostrar os resultados para o usuário usando componentes do Angular e muitos outros recursos.

Instalação da CLI Angular e início de um novo projeto

Primeiro, vamos instalar o **Angular CLI (Angular Command Line Interface)** que nos ajudará muito com a preparação e desenvolvimento em geral do projeto **Angular**.

Para instalar o Angular CLI, precisamos abrir o terminal integrado do Visual Studio Code ou um prompt de comando do sistema operacional e então executar o comando abaixo:

```
npm install -g @angular/cli
```

Em computadores Windows, a execução de scripts do **PowerShell** é desabilitada por padrão. Para permitir a execução de **scripts do PowerShell**, que é necessário para os binários globais do **npm**, você deve definir a seguinte política de execução, com o comando abaixo:

```
Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy RemoteSigned
```

Leia atentamente a mensagem exibida após a execução do comando e siga as instruções. Certifique-se de entender as implicações da definição de uma diretiva de execução.

Para testar a instalação e/ou checar a versão instalada use o comando:

```
ng version
```

Caso a versão instalada seja anterior a versão 13, você pode remover esta versão com os comandos abaixo, para fazer uma instalação atualizada

```
npm uninstall -g @angular/cli  
npm cache clean --force
```

Após remover a versão anterior, utilize o comando mencionado no começo para realizar a instalação da versão mais recente.

Após a instalação do **CLI Angular**, podemos criar um projeto **Angular SPA** na pasta **frontend** de nosso projeto, acesse a pasta e execute o comando abaixo:

```
ng new AccountOwnerClient --strict false
```

Duas questões serão exibidas. A primeira é se queremos que nosso projeto tenha roteamento criado, e a segunda é sobre que tipo de estilo queremos em nosso projeto. Vamos responder **Sim (Y)** para a primeira pergunta, e (**CSS** – basta apertar **enter**) para a segunda.

Levará algum tempo para criar o projeto.

Bibliotecas de terceiros como parte da preparação do projeto Angular

Vamos usar a biblioteca **ngx-bootstrap** (**angular bootstrap**), para estilização do projeto, para instalá-la abra o terminal, acesse a pasta do projeto **frontend** e execute os comandos abaixo:

```
cd AccountOwnerClient
ng add ngx-bootstrap
```

Será exibida uma mensagem informando que será realizada a instalação do **ngx-bootstrap**, e solicitando uma confirmação, pressione **Y**, para continuar com a instalação.

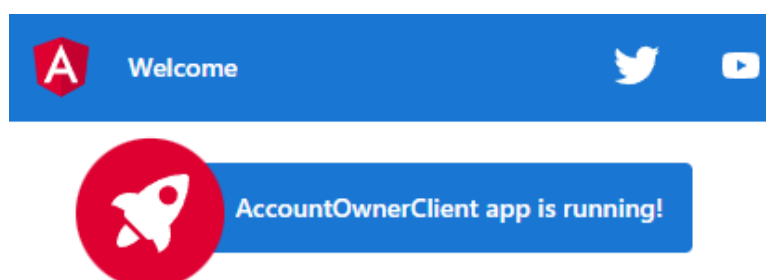
Com a instalação concluída, será instalado o **bootstrap** e o **ngx-bootstrap**, e também serão atualizados os arquivos **package.json**, **angular.json** e **app.module**.

Para rodar sua aplicação execute o comando (a **flag -o** abre automaticamente o navegador):

```
ng serve -o
```

Se ocorrer um erro de execução (isso pode acontecer devido a novas versões do **ngx-bootstrap**), abra o arquivo **angular.json** e faça a modificação em destaque para corrigir o caminho do **ngx-bootstrap**:

```
"styles": [
  "./node_modules/ngx-bootstrap/datepicker/bs-datepicker.css",
  "./node_modules/bootstrap/dist/css/bootstrap.min.css",
  "src/styles.css"
],
```



Para encerrar a aplicação pressione no terminal **Ctrl + C**.

Criando Componentes do Angular

O **Angular** é um **framework** para criação de aplicações **SPA (Single Page Application)**. Portanto, vamos gerar todas as nossas páginas dentro de uma única página. É por isso que só temos a página **index.html** dentro da pasta **src**. No arquivo **index.html** todo o conteúdo será gerado dentro da tag **<app-root></app-root>**, que vem do arquivo **app.component.ts**.

Dito isso, vamos olhar o arquivo **src/app/app.component.ts**:

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'AccountOwnerClient';
}
```

Todo componente deve importar da classe **Component** do pacote **@angular/core**. Iremos importar mais coisas quando precisarmos. Além disso, podemos notar que o decorador **@Component** dentro do código.

Este é o lugar onde criamos nosso seletor (é o mesmo que a tag **app-root** no arquivo **index.html**). Além disso, estamos vinculando o modelo **HTML** para este componente com o **templateUrl** e os arquivos **CSS** com este componente usando o **stylesUrls**. **StyleUrls** é uma vetor de cadeias de caracteres, separadas por vírgula.

Por fim, temos a classe exportada para o componente.

Agora vamos dar uma olhada no arquivo **app.module.ts**, que é bastante importante para a preparação do projeto **Angular** e para todo o desenvolvimento:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    BrowserAnimationsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Neste arquivo, vamos importar os módulos, componentes e serviços necessários. Vamos usar o vetor de declarações para importar nossos componentes e o vetor **imports** para importar nossos módulos. Além disso, vamos usar o vetor **providers** para registrar nossos serviços.

Criando Um Componente no Angular

Para criar um componente com o nome **Home**, execute o comando abaixo:

```
ng g component home --skip-tests
```

Com esse comando, criamos o componente **Home** com três arquivos (**.ts**, **.html**, **.css**) e atualizamos o arquivo **app.module**.

Além disso, ao adicionar o sinalizador **--skip-tests**, impedimos a criação do arquivo de teste.

Após a criação, podemos inspecionar o componente **Home**, e adicionar:

Abra o arquivo **home.component.ts** localizado no caminho **AccountOwnerClient\src\app\home** e faça as alterações abaixo:

```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-home',
  templateUrl: './home.component.html',
  styleUrls: ['./home.component.css']
})
export class HomeComponent implements OnInit {

  constructor() { }

  ngOnInit(): void {
  }
}
```

Aqui nós importamos **interface OnInit** que define a função **ngOnInit**. Essa função executará qualquer lógica dentro dela assim que o componente for inicializado.

Podemos notar o método construtor também. O construtor destina-se apenas à injeção do serviço no componente. Para qualquer ação que precise ser executada após a inicialização do componente, devemos usar a função **ngOnInit**.

Sobre o App.Module

Se verificarmos o arquivo **app.module.ts**, veremos que nosso novo componente é importado com o comando anterior:

```

import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModuleModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
import { HomeComponent } from './home/home.component';

@NgModule({
  declarations: [
    AppComponent,
    HomeComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModuleModule,
    BrowserAnimationsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

Mesmo que um módulo seja suficiente para toda a aplicação, ainda iremos criar mais módulos. Por quê?

Porque é mais fácil para manutenção dos módulos e além de nos dar uma vantagem no uso do **lazy content loading**. Isso significa que nosso aplicativo carregará apenas o conteúdo relacionado ao módulo específico para o qual estamos apontando, e não o aplicativo inteiro.

O **Lazy Content Load** (também conhecido como **Lazy Content Loading**, ou “carregamento preguiçoso de conteúdo”, em bom português) é uma técnica de otimização de desempenho de páginas que tem como objetivo fazer com que conteúdos mais pesados sejam carregados de forma assíncrona ou condicional. Isto é, **lazy load**, é fazer com que vídeos e imagens carreguem somente depois que o conteúdo acima da dobra seja totalmente carregado, ou quando for exibido.

Conteúdo adicional no componente Home

Vamos modificar o arquivo **home.component.ts**:

```

export class HomeComponent implements OnInit {

  public homeText: string;

  constructor() { }

  ngOnInit(): void {
    this.homeText = "BEM VINDO A APLICAÇÃO ACCOUNT-OWNER";
  }
}

```

Em seguida, vamos adicionar uma classe **CSS** ao arquivo **home.component.css**:

```
.homeText{
  font-size: 35px;
  color: red;
  text-align: center;
  position: relative;
  top: 30px;
  text-shadow: 2px 2px 2px gray;
}
```

Para continuar, vamos alterar o arquivo **home.component.html**:

```
<p class="homeText">{{homeText}}</p>
```

Finalmente, vamos modificar o arquivo **app.component.html**, removendo todo o conteúdo e adicionando um novo, apenas para testar se funciona:

```
<div class="container">
  <div class="row">
    <div class="col">
      <app-home></app-home>
    </div>
  </div>
</div>
```

Salve todos os arquivos e no terminal, vamos digitar novamente o comando de execução abaixo e aguardar que o aplicativo compile e execute. Devemos ver a mensagem de boas-vindas do **componente Home** no navegador.

```
ng serve -o
```



Neste momento, temos um componente funcionando e um aplicativo Angular que podemos executar em nosso navegador. Mas é só o começo. Temos um longo caminho pela frente porque ainda há muitos recursos importantes do Angular para introduzir no projeto.

Navegação e Roteamento no Angular

Um dos principais recursos de uma aplicação Web é sua navegação e, para habilitá-lo em nosso projeto, precisamos usar o roteamento. O **Angular Router** permite a navegação de uma exibição (**view**) para a próxima à medida que os usuários executam tarefas na aplicação.

Em nosso menu de navegação, teremos três opções: uma para a tela inicial, outra para as operações do proprietário e a última para as operações da conta. Com isso você poderá perceber as vantagens de usar vários módulos dentro de um projeto e como **Lazy Content Loading (carregamento de conteúdo preguiçoso)** ajuda nosso aplicativo a ter um melhor desempenho.

Criando um Menu de Navegação

Então, vamos começar criando um componente com o nome **Menu**. No terminal execute o comando:

```
ng g component menu --skip-tests
```

Vamos usar as classes do **Bootstrap** para implementar o menu de navegação dentro do arquivo **menu.component.html**, conforme código abaixo:

```
<div class="row">
  <div class="col">
    <nav class="navbar navbar-expand-lg navbar-dark bg-dark">
      <div class="container-fluid">
        <a class="navbar-brand" href="#">Account-Owner Home</a>
        <button class="navbar-toggler" type="button" data-bs-toggle="collapse"
data-bs-target="#collapseNav"
        aria-controls="collapseNav" aria-expanded="false" aria-label="Toggle navigation">
          <span class="navbar-toggler-icon"></span>
        </button>

        <div class="collapse navbar-collapse" id="collapseNav">
          <ul class="navbar-nav me-auto mb-2 mb-lg-0">
            <li class="nav-item">
              <a class="nav-link" href="#">Owner Actions </a>
            </li>
            <li class="nav-item">
              <a class="nav-link" href="#">Account Actions </a>
            </li>
          </ul>
        </div>
      </div>
    </nav>
  </div>
</div>
```

Por enquanto não vamos modificar o arquivo **menu.component.ts**.

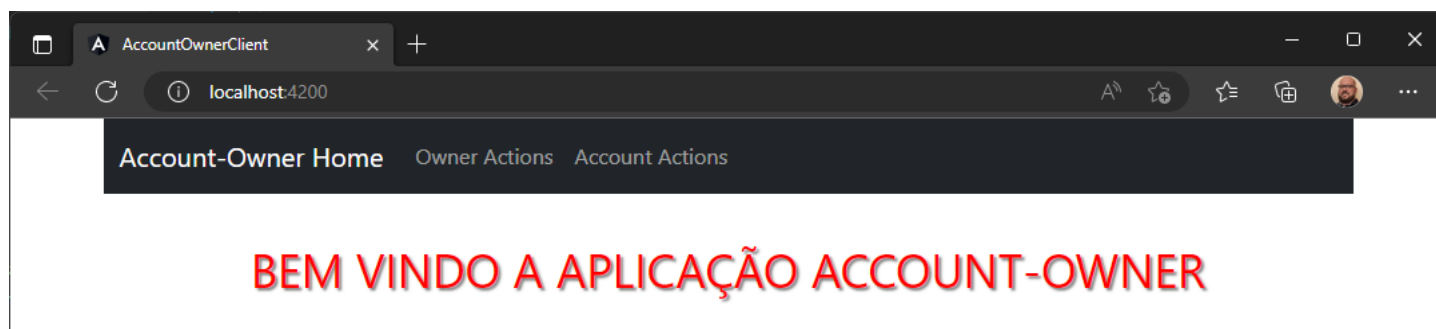
Mas, vamos mudar o arquivo **app.component.html**, para adicionar o menu criado:

```
<div class="container">
  <div class="row">
    <div class="col">
      <app-menu></app-menu>
    </div>
  </div>
  <div class="row">
    <div class="col">
      <app-home></app-home>
    </div>
  </div>
</div>
```

Salve todos os arquivos e no terminal execute o comando de execução da aplicação para iniciar o projeto:

```
ng serve -o
```

Assim que o projeto é executado, veremos nosso menu na tela:



Adicionando funcionalidade de recolhimento à NavBar

Agora, se encolhermos nossa tela, poderemos ver o botão de menu, e uma vez que clicamos nele, deveria mostrar nossos itens de menu. Mas isso não acontece porque não instalamos todas as partes do **JavaScript** para o **Bootstrap**, e nem queremos isso.

O que queremos é usar o **ngx-bootstrap** o máximo que pudermos com todos os componentes que ele nos fornece.

Dito isto, vamos usar o componente **Collapse** do **ngx-bootstrap** para ativar nosso menu recolhível.

Então, vamos primeiro importar o componente no arquivo **app.module.ts**, através das alterações abaixo:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
import { HomeComponent } from './home/home.component';
```



```
import { MenuComponent } from './menu/menu.component';
import { CollapseModule } from 'ngx-bootstrap/collapse';

@NgModule({
  declarations: [
    AppComponent,
    HomeComponent,
    MenuComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    BrowserModuleAnimationsModule,
    CollapseModule.forRoot()
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Neste arquivo, também podemos notar um **MenuComponent** dentro do vetor **declarations**. Ele foi importado anteriormente durante a criação do componente de navegação.

Em seguida, vamos adicionar uma propriedade dentro do arquivo **menu.component.ts**:

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-menu',
  templateUrl: './menu.component.html',
  styleUrls: ['./menu.component.css']
})
export class MenuComponent {
  isCollapsed: boolean = false;
}
```

E, finalmente, vamos modificar nosso código HTML da NavBar. Abra o arquivo **menu.component.html** localizado no caminho **AccountOwnerClient\src\app\menu** e faça as alterações em destaque:

```
<div class="row">
  <div class="col">
    <nav class="navbar navbar-expand-lg navbar-dark bg-dark">
      <div class="container-fluid">
        <a class="navbar-brand" href="#">Account-Owner Home</a>
        <button class="navbar-toggler" type="button" (click)="isCollapsed = !isCollapsed"
          [attr.aria-expanded]="!isCollapsed" aria-controls="collapseNav">
          <span class="navbar-toggler-icon"></span>
        </button>

        <div class="collapse navbar-collapse" id="collapseNav" [collapse]="!isCollapsed"
          [isAnimated]="true">
```

```

<ul class="navbar-nav me-auto mb-2 mb-lg-0">
  <li class="nav-item">
    <a class="nav-link" href="#">Owner Actions </a>
  </li>
  <li class="nav-item">
    <a class="nav-link" href="#">Account Actions </a>
  </li>
</ul>
</div>
</div>
</nav>
</div>
</div>

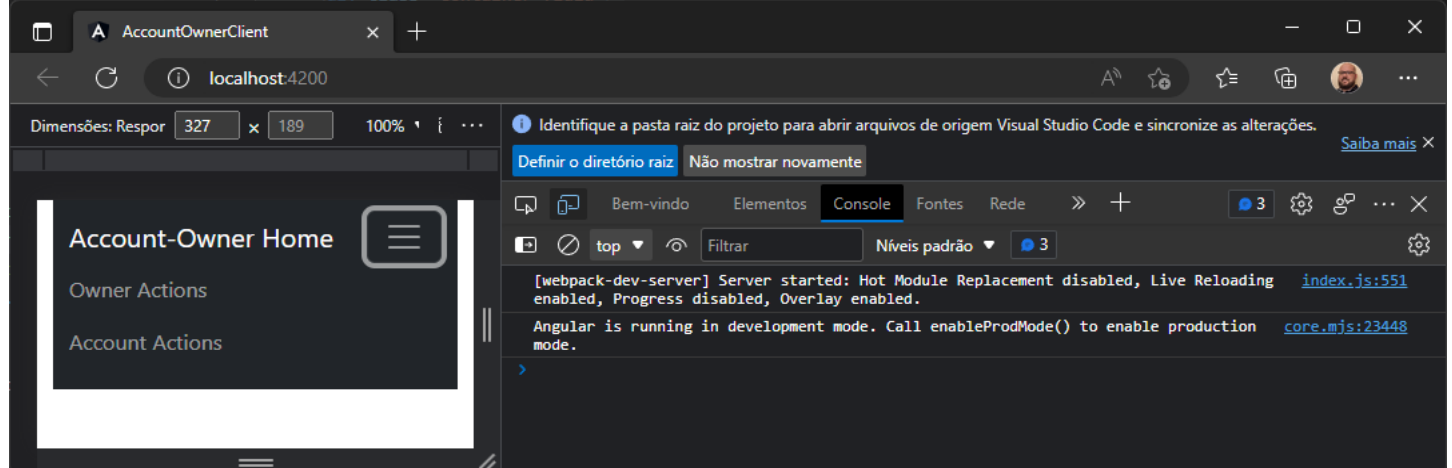
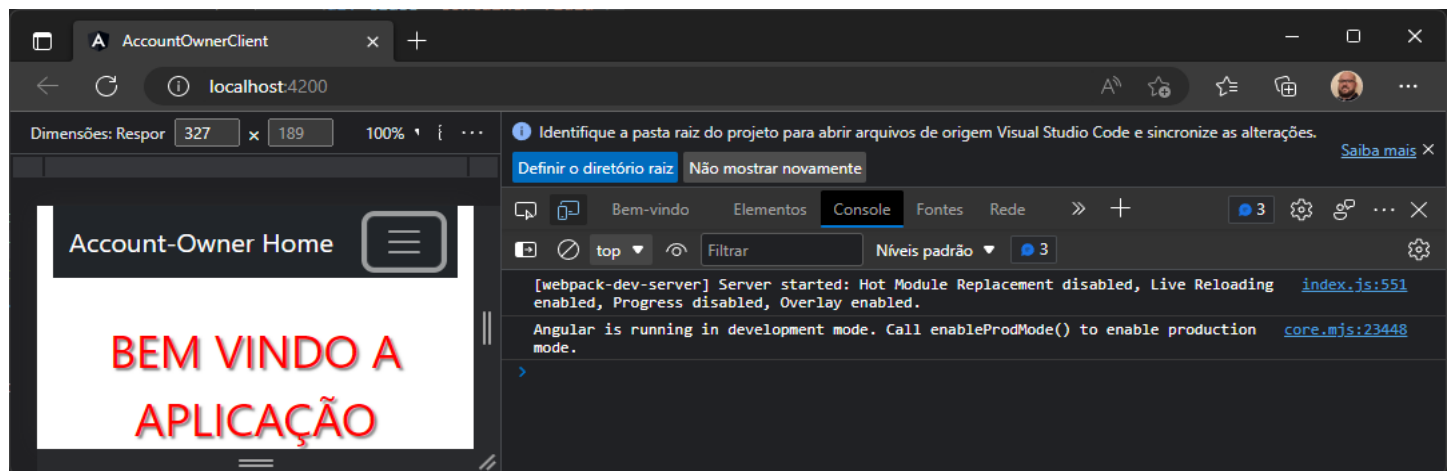
```

Aqui adicionamos o evento (**click**), que mudará o estado da propriedade **isCollapsed** cada vez que clicarmos no botão de menu. Além disso, preste atenção que o atributo **aria-controls** deve ter o mesmo valor do **id** que **div** abaixo dele. Além disso, na **div** mencionada, definimos o valor do seletor **[collapse]** para indicar a visibilidade do nosso conteúdo e definimos a animação como **true** usando o input **[isAnimated]**.

E só isso. Agora podemos encolher a tela e, uma vez que o botão de menu aparece, podemos clicar nele e ver nossos itens de menu.

Salve todos os arquivos e no terminal execute o comando de execução da aplicação, conferindo se o caminho apresentado é da pasta **frontend\AccountOwnerClient**, para testes:

```
ng serve -o
```



Configurando o Roteamento do Angular

Para habilitar a navegação entre todas as páginas dentro do projeto, precisamos configurar o roteamento no Angular. O módulo de roteamento já foi criado para nós, pois o solicitamos durante a criação do projeto.

Podemos ver isso no arquivo **app.module.ts**:

```
import { AppRoutingModule } from './app-routing.module';

...

imports: [
  BrowserModule,
  AppRoutingModule,
  BrowserModuleAnimationsModule,
  CollapseModule.forRoot()
],
```

O **AppRoutingModule** é o módulo responsável pelo roteamento no angular.

Agora, tudo o que temos a fazer é modificar o arquivo **app-routing.module.ts**:

```
import { HomeComponent } from './home/home.component';

import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';

const routes: Routes = [
  { path: 'home', component: HomeComponent },
  { path: '', redirectTo: '/home', pathMatch: 'full' }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Adicionamos duas rotas dentro do vetor **routes**. Esse vetor já é fornecido dentro da função para definir rotas para nossa aplicação.

Quando criamos mais de um módulo dentro do aplicativo, podemos usar a função **forRoot()** fornecida pelo **RouterModule**, somente no módulo **main(root)**. Em todos os outros módulos, devemos usar a função **forChild()**. A função **forRoot()** aceita um vetor de objetos como parâmetro. Cada elemento desse vetor consiste no caminho e no componente de destino dessa rota. Então, o caminho: **home** significa que no endereço **http://localhost/4200/home**, vamos servir o **HomeComponent**. A outra linha dentro do vetor **routes** é o redirecionamento padrão para a home page.

Agora, para habilitar o conteúdo das rotas, temos que modificar o arquivo **app.component.html**:

```

<div class="container">
  <div class="row">
    <div class="col">
      <app-menu></app-menu>
    </div>
  </div>
  <div class="row">
    <div class="col">
      <router-outlet></router-outlet>
    </div>
  </div>
</div>

```

O **router-outlet** é um contêiner para o conteúdo de roteamento. Então, basicamente, todo o conteúdo que existe no endereço para o qual estamos roteando será apresentado dentro desse contêiner.

Agora, se navegarmos para **localhost:4200** devemos ser capazes de ver o mesmo resultado de antes, mas desta vez, estamos fornecendo nosso componente **home** através **<router-outlet>** e não pelo **<app-home>**.

Além disso, se clicarmos em qualquer outro item do menu, seremos automaticamente redirecionados para a página inicial.

Estilizando links

Se quisermos estilizar seu **link ativo** no **menu**, temos que alterar nossa **tag <a>**. Abra o arquivo **menu.component.html** localizado no caminho **AccountOwnerClient\src\app\menu** e faça as alterações em destaque:

```

<div class="row">
  <div class="col">
    <nav class="navbar navbar-expand-lg navbar-dark bg-dark">
      <div class="container-fluid">
        <a class="navbar-brand" [routerLink]="['/home']" routerLinkActive="active"
          [routerLinkActiveOptions]="{exact: true}">Account-Owner Home</a>
        <button class="navbar-toggler" type="button" (click)="isCollapsed = !isCollapsed"
          [attr.aria-expanded]="!isCollapsed" aria-controls="collapseNav">
          <span class="navbar-toggler-icon"></span>
        </button>

        <div class="collapse navbar-collapse" id="collapseNav" [collapse]="!isCollapsed"
[isAnimated]="true">
          <ul class="navbar-nav me-auto mb-2 mb-lg-0">
            <li class="nav-item">
              <a class="nav-link" href="#">Owner Actions </a>
            </li>
            <li class="nav-item">
              <a class="nav-link" href="#">Account Actions </a>
            </li>
          </ul>
        </div>
      </div>
    </nav>
  </div>

```

```
</div>
</nav>
</div>
</div>
```

Com o **routerLinkActive**, estamos configurando o nome da classe **CSS** que queremos usar para estilizar o link ativo. Além disso, o **routerLinkActiveOptions** vai nos permitir adicionar uma classe somente se houver uma correspondência exata do link e da **URL**. Por fim, não estamos mais usando o atributo para **href** para navegação. Em vez disso, estamos usando a diretiva **[routerlink]** para navegar até o nosso caminho de roteamento.

Agora no arquivo **menu.component.css** faça as alterações abaixo para adicionar a classe **CSS .active**:

```
.active{
  font-weight: bold;
  font-style: italic;
  color: #fff;
}
```

Excelente. Se inspecionarmos nosso aplicativo, veremos que o link **Account-Owner Home** agora é branco e está em negrito.

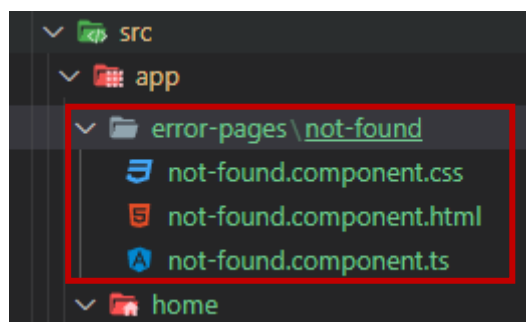
Criando o componente Not-Found (Não Encontrado)

Temos navegação funcionando. Para completar a parte de roteamento do Angular deste projeto, vamos criar um componente com o nome **not-found**. A aplicação vai redirecionar o usuário para esse componente quando ele digitar uma rota inexistente na **URL**.

Para fazer isso, vamos executar no terminal o comando:

```
ng g component error-pages/not-found --skip-tests
```

O resultado do comando e a estrutura de pasta criada na figura abaixo:



Vamos modificar o arquivo **not-found.component.ts**:

```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-not-found',
  templateUrl: './not-found.component.html',
```

```

    styleUrls: ['./not-found.component.css']
  })
  export class NotFoundComponent implements OnInit {
    notFoundText: string = `404 SORRY COULDN'T FIND IT!!!`;

    constructor() { }

    ngOnInit(): void {
    }
  }
}

```

Temos que prestar atenção ao valor da cadeia de caracteres da propriedade **notFoundText**. Não estamos usando apóstrofes, mas sim crases ('). Todo o conteúdo dentro das crases será considerado como uma **string**, até mesmo o sinal de apóstrofo no texto.

Para continuar, vamos modificar o arquivo **not-found.component.html**, faça as alterações em destaque:

```

<p>
  {{notFoundText}}
</p>

```

Além disso, precisamos modificar o arquivo **not-found.component.css**, faça as alterações em abaixo:

```

p {
  font-weight: bold;
  font-size: 50px;
  text-align: center;
  color: #f10b0b;
}

```

E finalmente, vamos alterar o conteúdo do vetor **routes** do arquivo **app-routing.module.ts**:

```

import { HomeComponent } from './home/home.component';

import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { NotFoundComponent } from './error-pages/not-found/not-found.component';

const routes: Routes = [
  { path: 'home', component: HomeComponent },
  { path: '404', component: NotFoundComponent },
  { path: '', redirectTo: '/home', pathMatch: 'full' },
  { path: '**', redirectTo: '/404', pathMatch: 'full' }
];

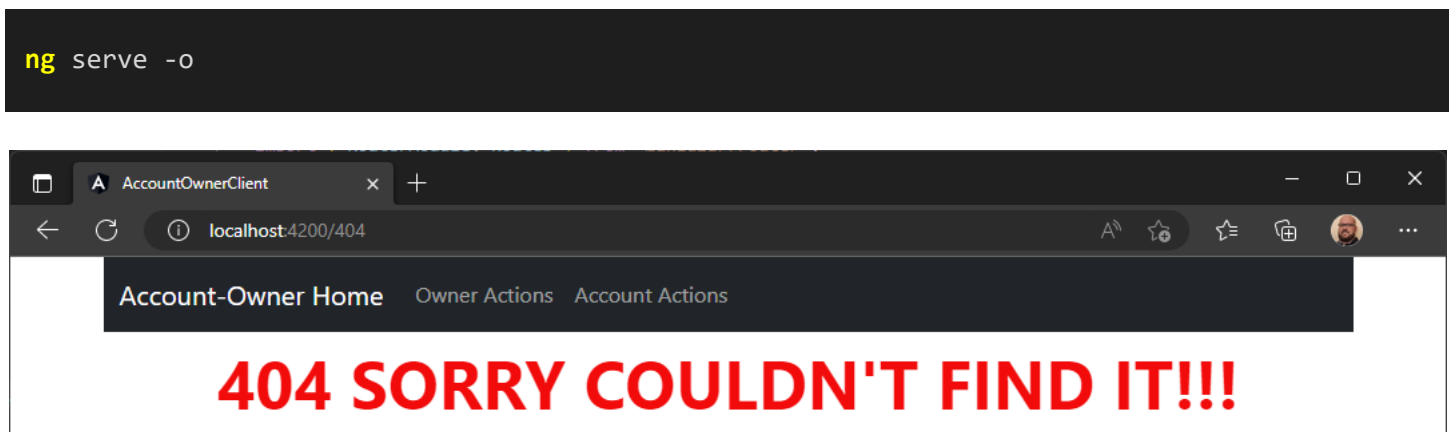
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }

```

Há duas mudanças aqui. Com a primeira alteração, declaramos o caminho **404** e atribuímos o componente **NotFoundComponent** a esse caminho. Agora nosso componente será visível no **localhost:4200/404**. A segunda alteração significa que, sempre que pesquisarmos qualquer rota que não corresponda a nenhuma de nossas rotas definidas, o aplicativo nos redireciona para a página **404**.

Digitando **localhost:4200/qualquerCoisa** no seu navegador será retornada a página não encontrada. Além disso, podemos tentar navegar para **localhost:4200/404**, e o aplicativo nos mostrará a mesma página.

Salve todos os arquivos e no terminal execute o comando de execução da aplicação, conferindo se o caminho apresentado é da pasta **frontend\AccountOwnerClient**, para testes:



Como você deve ter notado, criar o menu e usar o roteamento em projetos do **Angular** é bastante simples. Embora não estejamos criando um projeto grande, ele é grande o suficiente para demonstrar o uso, a configuração e o roteamento de todas as páginas que temos atualmente. Claro, vamos criar rotas para todas as novas páginas que introduzimos em nosso projeto.

HttpClient, Services e Arquivos de Ambiente do Angular

Ao enviar solicitações HTTP para o nosso servidor, precisamos usar o **Angular HttpClient**. É claro que podemos lidar com todas as solicitações **HTTP** de todos os componentes e processar a resposta também, mas não é uma boa prática. É muito melhor criar um repositório para suas solicitações e, em seguida, enviar o **URI** da solicitação para esse repositório. O repositório deve cuidar do resto.

Então, vamos começar com os arquivos de ambiente primeiro.

Trabalhando com arquivos de ambiente

Enquanto este projeto está no modo de desenvolvimento, o endereço de **endpoint** do servidor é o **http://localhost:[porta]**. Com o aplicativo no ambiente de produção, o **endpoint** é diferente, algo como **www.accountowner.com**. Então, queremos que o Angular cuide disso. Dito isto, no modo de desenvolvimento, ele deve enviar solicitações para o **URI** de desenvolvimento e, no ambiente de produção, para um endereço diferente.

Vamos implementar isso.

Nas versões mais recentes do Angular, devemos usar um comando no terminal para criar os arquivos de configuração de ambiente, desta forma execute o comando abaixo:

```
ng generate environments
```

Na janela do explorador do **Visual Studio Code**, vamos procurar a pasta de ambientes criada: **src/environments**. Dentro dessa pasta, vamos encontrar dois arquivos, o **environment.ts**, e o **environment.development.ts**. Vamos usar o primeiro para a configuração do ambiente de produção (quando sua aplicação já está online e em uso) e o segundo para a configuração do ambiente de desenvolvimento.

Então, vamos começar com as modificações no arquivo **environment.ts**:

```
export const environment = {  
  production: true,  
  urlAddress: 'http://www.accountowner.com'  
};
```

Em seguida, vamos modificar o arquivo **environment.development.ts**:

```
export const environment = {  
  production: false,  
  urlAddress: 'http://localhost:5000'  
};
```

Agora vamos criar um serviço, que poderemos usar para obter o ambiente válido da **urlAddress**.

Abra o terminal e execute o comando abaixo para o arquivo de serviço dentro de uma nova pasta compartilhada para os demais serviços:


```
ng g service shared/services/environment-url --skip-tests
```

Sobre os Serviços no Angular (Angular Services)

Os serviços são apenas classes, que nos fornecem alguma lógica de negócios relevante para nossos componentes. Esses serviços devem ser injetados em um componente usando injeção de construtor. Além disso, nosso código se torna mais fácil de manter e legível quando extraímos a lógica de um componente para o serviço.

Quando queremos usar um serviço, precisamos injetá-lo no construtor de um componente. Por isso, é sempre decorado com o decorador **@Injectable**.

Devemos usar serviços sempre que tivermos código que possamos reutilizar em outros componentes ou extrair parte do código de nossos componentes.

Dito isto, vamos continuar nosso trabalho com os arquivos de ambiente modificando o arquivo **environment-url.service.ts**, criado anteriormente.

Abra o arquivo `\src\app\shared\services\environment-url.service.ts`, e faça as alterações em destaque:

```
import { environment } from './../../../../environments/environment';
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class EnvironmentUrlService {
  urlAddress: string = environment.urlAddress;

  constructor() { }
}
```

A propriedade **urlAddress** aceita o valor de **urlAddress** definido no arquivo de ambiente. O Angular sabe se é um ambiente de produção ou desenvolvimento e vai retornar um valor válido para a **urlAddress**. Podemos verificar isso no arquivo **angular.json**:

```
"fileReplacements": [
  {
    "replace": "src/environments/environment.ts",
    "with": "src/environments/environment.development.ts"
  }
]
```

Podemos ver que o Angular substituirá os arquivos no modo de desenvolvimento.

Criando um Arquivo de Repositório Angular

Agora podemos configurar o **HttpClientModule** e criar o serviço de repositório.

Primeiro, precisamos importar o **HttpClientModule** dentro do arquivo **src\app\app.module.ts**:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HttpClientModule } from '@angular/common/http';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
import { HomeComponent } from './home/home.component';
import { MenuComponent } from './menu/menu.component';
import { CollapseModule } from 'ngx-bootstrap/collapse';
import { NotFoundComponent } from './error-pages/not-found/not-found.component';

@NgModule({
  declarations: [
    AppComponent,
    HomeComponent,
    MenuComponent,
    NotFoundComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    BrowserAnimationsModule,
    HttpClientModule,
    CollapseModule.forRoot()
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Depois disso, vamos criar um arquivo de serviço e nomeá-lo **owner-repository.service.ts**. Vamos colocá-lo na mesma pasta em que do serviço de ambiente. Neste serviço, vamos criar solicitações **GET**, **POST**, **PUT** e **DELETE** para a entidade **owner** a partir de nossa **API**.

Abra o terminal e execute o comando abaixo para o arquivo de repositório:

```
ng g service shared/services/owner-repository --skip-tests
```

Agora crie uma pasta com o nome **_interfaces** dentro da pasta app, o caminho completo do local de criação da pasta é **AccountOwnerClient\src\app**.

Dentro da pasta **_interfaces** adicione um arquivo com o nome **owner.model.ts**, e faça a codificação abaixo:

```
export interface Owner {
  id: string;
  name: string;
  dateOfBirth: Date;
  address: string;
}
```

Uma vez que tenhamos nossa interface no lugar, podemos modificar o arquivo do repositório. Abra o arquivo `\src\app\shared\services\owner-repository.service.ts` e faça as alterações abaixo:

```
import { Owner } from '../../../_interfaces/owner.model';
import { EnvironmentUrlService } from './environment-url.service';
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class OwnerRepositoryService {

  constructor(private http: HttpClient, private envUrl: EnvironmentUrlService) { }

  public getOwners = (route: string) => {
    return this.http.get<Owner[]>(this.createCompleteRoute(route, this.envUrl.urlAddress));
  }

  public createOwner = (route: string, owner: Owner) => {
    return this.http.post<Owner>(this.createCompleteRoute(route, this.envUrl.urlAddress),
owner, this.generateHeaders());
  }

  public updateOwner = (route: string, owner: Owner) => {
    return this.http.put(this.createCompleteRoute(route, this.envUrl.urlAddress), owner,
this.generateHeaders());
  }

  public deleteOwner = (route: string) => {
    return this.http.delete(this.createCompleteRoute(route, this.envUrl.urlAddress));
  }

  private createCompleteRoute = (route: string, envAddress: string) => {
    return `${envAddress}/${route}`;
  }

  private generateHeaders = () => {
    return {
      headers: new HttpHeaders({ 'Content-Type': 'application/json' })
    }
  }
}
```

Entendendo o Código do Repositório

Primeiro, injetamos o **Angular HttpClient** e a variável de ambiente no construtor. Em seguida, criamos funções que vão suprir nossas solicitações. A função **getOwners** é um **wrapper** para a solicitação **GET**. Ele aceita o parâmetro **route** do tipo de **string (api/owner)** e, em seguida, combina-o com a variável de ambiente (**localhost** ou **www...**). Depois de tudo isso, teremos uma rota como **http://localhost:5000/api/owner** se for no ambiente de desenvolvimento, e que se encaixa perfeitamente nos requisitos no lado do servidor.

A segunda função, **createOwner**, é um **wrapper** para uma solicitação **POST**. Ele também gera uma rota, mas também recebe um corpo (uma entidade que estamos criando) e gera cabeçalhos. Para este exemplo, estamos apenas criando o **Content-Type** dentro do cabeçalho. Mas se precisarmos de valores adicionais dentro do cabeçalho, poderíamos simplesmente adicionar outro par de chave-valor dentro do objeto **HttpHeaders**.

A função **updateOwner** é praticamente a mesma que a função **create**, exceto que ela envia a solicitação **PUT**. Por fim, a função **deleteOwner** é um **wrapper** para a solicitação **DELETE** que aceita uma rota como (**api/owner/id**). Ambas as funções estão faltando um método **HTTP** fortemente tipado (**put** e **delete**) porque não esperamos nenhum objeto como resposta do servidor. Se tudo correr bem, vamos receber um **StatusCode 204** como resposta sem um corpo (dados).

A Assinatura nas chamadas HTTP

Essas funções de **wrapper** precisam de uma assinatura para funcionar. Até este ponto, estamos apenas criando um repositório com as chamadas HTTP. Mas assim que começarmos a criar nossas páginas, usaremos a assinatura correspondente a cada um.

Por enquanto, vamos apenas mostrar-lhe um exemplo de uma assinatura:

```
owners: Owner[];

constructor(private repo: OwnerRepositoryService) { }

private consumeGetFromRepository = () => {
  this.repo.getOwners('api/owner')
    .subscribe(own => {
      this.owners = own;
    })
}
```

Como você pode notar, estamos chamando a função **getOwners** do repositório, mas essa função não será executada até que chamemos a função **subscribe**. O resultado da resposta será armazenado no parâmetro **own**.

Angular Lazy Loading

Como continuação da última parte (onde vimos sobre as assinaturas), agora implementaremos essa assinatura em nossas solicitações **HTTP** para exibir os dados na página. Além disso, vamos usar a vantagem do **Angular Lazy Loading (Carregamento Preguiçoso)**, usando outro módulo em nossa aplicação – o módulo do **owner**.

Criando um Novo Módulo

Vamos começar pela criação do módulo **owner**, através das linhas de comando do **Angular CLI**, no terminal execute o comando abaixo:

```
ng g module owner --routing=true --module app.module
```

Este comando faz várias coisas. Ele cria um módulo **Owner**, também cria um arquivo de roteamento para esse módulo e, finalmente, atualiza o arquivo do **app.module.ts**:

```
CREATE src/app/owner/owner-routing.module.ts (248 bytes)
CREATE src/app/owner/owner.module.ts (276 bytes)
UPDATE src/app/app.module.ts (989 bytes)
```

Vamos dar uma olhada no arquivo **owner.module.ts**:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

import { OwnerRoutingModule } from './owner-routing.module';

@NgModule({
  declarations: [],
  imports: [
    CommonModule,
    OwnerRoutingModule
  ]
})
export class OwnerModule { }
```

Há duas pequenas diferenças entre esse arquivo de módulo e o arquivo **app.module** da aplicação. A primeira diferença é que no arquivo de módulo do aplicativo temos uma instrução **import** para o **BrowserModule**, e no arquivo do módulo **owner**, temos uma instrução **import** para o **CommonModule**. Isso porque o **BrowserModule** está relacionado apenas ao módulo raiz da aplicação.

A segunda diferença é que não temos o vetor **providers** no arquivo de módulo **owner**. Isso porque devemos registrar todos os serviços no módulo raiz. Dessa forma, os componentes injetarão a mesma instância do serviço apenas uma vez e você poderá manter o estado em seu serviço.

É claro que, se realmente quisermos registrar um serviço dentro de qualquer módulo filho, podemos simplesmente adicionar o vetor **providers**. Mas, ao fazer isso, não podemos manter o estado dentro do nosso serviço, porque toda vez que criamos uma instância desse componente, uma nova instância do serviço é criada.

Finalmente, podemos ver que este módulo importa o **OwnerRoutingModule** de um arquivo separado.

Componente Owner e o Angular Lazy Loading

Vamos começar com a criação dos arquivos do componente **owner**

Abra o terminal e execute o comando abaixo para criar os arquivos do componente **owner-list**:

```
ng g component owner/owner-list --skip-tests
```

Este comando vai criar a estrutura de pastas necessária e vai importar este componente dentro do arquivo também **owner.module.ts**.

O que queremos agora é que, quando clicamos no menu "**Owner-Actions**", seja exibido o conteúdo do arquivo **HTML** deste componente. Então, primeiro, apenas para fins de teste, vamos modificar o arquivo **owner.component.html** adicionando um parágrafo (**<p> tag**):

```
<p> This is owner-list componente page. </p>
```

Após essa alteração, abra o arquivo **\src\app\app-routing.module.ts** e faça as alterações em destaque:

```
const routes: Routes = [
  { path: 'home', component: HomeComponent },
  { path: 'owner', loadChildren:()=> import('./owner/owner.module').then(m =>m.OwnerModule)},
  { path: '404', component: NotFoundComponent },
  { path: '', redirectTo: '/home', pathMatch: 'full' },
  { path: '**', redirectTo: '/404', pathMatch: 'full' }
];
```

Com a parte modificada do código, estamos configurando o **app-routing.module** para carregar o módulo **owner** sempre que alguém procurar pelo **endpoint** **http://localhost:4200/owner**. Como podemos notar, estamos usando a propriedade **loadChildren**, o que significa que o módulo **owner** com seus componentes não será carregado até que os solicitemos explicitamente. Ao fazer isso, estamos configurando o **lazy content loading** do **Angular** a partir do conteúdo do módulo **owner**.

Agora, se navegarmos para a página inicial, obteremos apenas os recursos do módulo raiz, não do módulo **owner**. E somente navegando até o menu **owner-actions**, carregaremos os recursos do módulo **owner** da aplicação. A partir da declaração anterior, podemos ver por que o **Lazy Load** do **Angular** é importante para aplicações **Angular**.

Roteamento para o Módulo Owner

Agora, para habilitar a navegação para o componente **OwnerList**, temos que modificar o **owner-routing.module.ts**:

Abra o arquivo `\src\app\owner\owner-routing.module.ts` e faça as alterações em destaque:

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';

import { OwnerListComponent } from '../owner-list/owner-list.component';

const routes: Routes = [
  { path: 'list', component: OwnerListComponent }
];

@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class OwnerRoutingModule { }
```

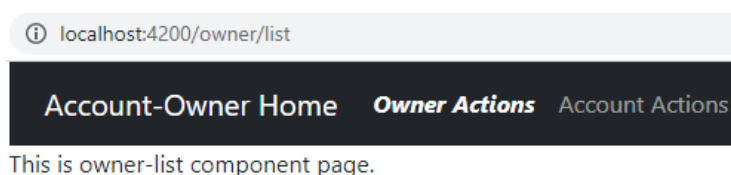
Com essa configuração, estamos expondo nosso **OwnerListComponent** através do **endpoint** **http://localhost:4200/owner/list**. Além disso, estamos usando a função **RouterModule.forChild** e não a função **forRoot**. Este é o caso porque devemos usar a função **forRoot** apenas no módulo raiz da aplicação.

Agora temos que modificar o arquivo **menu.component.html**.

Abra o arquivo `\src\app\menu\menu.component.html` e faça as alterações em destaque:

```
<div class="collapse navbar-collapse" id="collapseNav" [collapse]="!isCollapsed"
[isAnimated]="true">
  <ul class="navbar-nav me-auto mb-2 mb-lg-0">
    <li class="nav-item">
      <a class="nav-link" [routerLink]="['/owner/list']" routerLinkActive="active"
        [routerLinkActiveOptions]="{exact: true}"> Owner Actions </a>
    </li>
    <li class="nav-item">
      <a class="nav-link" href="#">Account Actions </a>
    </li>
  </ul>
</div>
```

Depois de todas essas modificações, podemos executar nossa aplicação e clicar no link **Owner Actions**. Assim que fizermos isso, nosso novo componente aparecerá e o link receberá um estilo de classe ativo.



Agora sabemos como configurar o roteamento para um módulo filho e para o componente dentro desse módulo também.

Assinatura e exibição de dados

Quando navegamos para o menu Ações do proprietário, queremos mostrar todos os proprietários ao usuário. Isso significa que quando o componente **owner** é carregado, o aplicativo obtém automaticamente todos os **owners** do servidor.

Já temos nossa interface **Owner** criada e vamos usá-la aqui.

Dito isto, vamos modificar o arquivo `\src\app\owner\owner-list\OwnerListComponent`:

```
import { Component, OnInit } from '@angular/core';

import { Owner } from '../../../_interfaces/owner.model';
import { OwnerRepositoryService } from '../../../shared/services/owner-repository.service';

@Component({
  selector: 'app-owner-list',
  templateUrl: './owner-list.component.html',
  styleUrls: ['./owner-list.component.css']
})
export class OwnerListComponent implements OnInit {
  owners: Owner[];

  constructor(private repository: OwnerRepositoryService) { }

  ngOnInit(): void {
    this.getAllOwners();
  }

  private getAllOwners = () => {
    const apiAddress: string = 'api/owner';
    this.repository.getOwners(apiAddress)
      .subscribe(own => {
        this.owners = own;
      })
  }
}
```

Temos a propriedade **owners** do tipo vetor de **Owner (Owner[])**. Em seguida, executamos a função de assinatura, que preencherá essa propriedade com todos os **owners** do servidor. Usando a propriedade **owners** para criar nossa página **HTML** é o que buscamos.

Para conseguir isso, vamos modificar o componente **HTML**:

Abra o arquivo `\src\app\owner\owner-list\owner-list.component.html` e faça a codificação abaixo:


```

<div class="row">
  <div class="offset-10 col-md-2 mt-2"> <a href="#">Create owner</a> </div>
</div> <br>
<div class="row">
  <div class="col-md-12">
    <div class="table-responsive">
      <table class="table table-striped">
        <thead>
          <tr>
            <th>Owner name</th>
            <th>Owner address</th>
            <th>Date of birth</th>
            <th>Details</th>
            <th>Update</th>
            <th>Delete</th>
          </tr>
        </thead>
        <tbody>
          <tr *ngFor="let owner of owners">
            <td>{{owner.name}}</td>
            <td>{{owner.address}}</td>
            <td>{{owner.dateOfBirth | date: 'dd/MM/yyyy'}}</td>
            <td><button type="button" id="details" class="btn btn-primary"> Details
              </button>
            </td>
            <td><button type="button" id="update" class="btn btn-success"> Update
              </button>
            </td>
            <td><button type="button" id="delete" class="btn btn-danger"> Delete
              </button>
            </td>
          </tr>
        </tbody>
      </table>
    </div>
  </div>
</div>

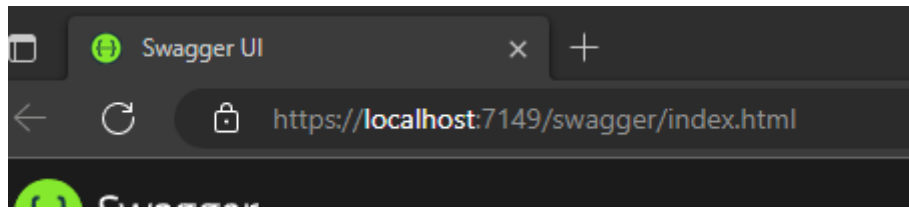
```

Usamos algumas classes básicas do **Bootstrap** para criar uma tabela mostrando os dados dos **owners**. Dentro dessa tabela, nós fazemos um **loop** sobre todos os **owners** com a diretiva ***ngFor**. Em seguida, usando a interpolação **{{ }}**, mostramos as propriedades do **owner** na página. Para a propriedade **dateOfBirth**, estamos usando apenas formatador **pipe** `| date: 'dd/MM/yyyy'` para formatá-la da maneira que queremos vê-la na página.

Agora podemos iniciar nosso servidor de dados, lembre-se que para isso, o banco de dados deve estar criado e em execução.

Abra um **novo terminal**, entre na pasta do **backend** e execute o projeto **AccountOwnerServer**:

```
dotnet watch run
```



Em seguida, com o servidor ainda rodando em um segundo terminal, abra o arquivo `\src\environment\environment.development.ts`, e altere a **linha 3** de acordo com o endereço apresentado no navegador, abaixo está o código de acordo com minha configuração, ou seja, com o número de porta exibida na imagem acima:

```
export const environment = {  
  production: false,  
  urlAddress: 'https://localhost:7149'  
};
```

Agora, salve todos os arquivos e no terminal (que não está rodando o servidor) execute o comando de execução da aplicação, conferindo se o caminho apresentado é da pasta **frontend\AccountOwnerClient**, para testes:

```
ng serve -o
```

A screenshot of a web browser showing the AccountOwnerClient application. The browser has two tabs: 'Swagger UI' and 'AccountOwnerClient'. The address bar shows 'localhost:4200/owner/list'. The application has a dark header with 'Account-Owner Home', 'Owner Actions' (highlighted), and 'Account Actions'. There is a link 'Create owner' in the top right. Below the header is a table with 6 columns: 'Owner name', 'Owner address', 'Date of birth', 'Details', 'Update', and 'Delete'. The table contains 6 rows of owner data. Each row has a 'Details' button (blue), an 'Update' button (green), and a 'Delete' button (red).

Owner name	Owner address	Date of birth	Details	Update	Delete
Anna Bosh	27 Colored Row	14/11/1974	Details	Update	Delete
John Keen	61 Wellfield Road	05/12/1980	Details	Update	Delete
José Antonio Gallo Junior	Rua Angelo Reginato, 44; Barra Bonita	05/08/1981	Details	Update	Delete
Martin Miller	3 Edgar Buildings	21/05/1983	Details	Update	Delete
Nick Somion	North sunny address 102	15/12/1998	Details	Update	Delete
Sam Query	91 Western Roads	22/04/1990	Details	Update	Delete

Tratamento de Erros no Angular

Ao enviar solicitações para o nosso servidor de API da Web, podemos obter um erro em resposta. Portanto, usar o tratamento de erros do Angular para lidar com esses erros durante o envio de solicitações HTTP é uma obrigação. É exatamente isso que vamos fazer neste post. Se obtivermos o erro 404 ou 500, vamos redirecionar o usuário para uma página específica. Para outros erros, vamos mostrar uma mensagem de erro em um formulário modal. A página que lida com o erro 404 já está criada, então, vamos continuar criando um componente 500 (Erro interno do servidor).

Criando o Componente de Erro Interno do Servidor

Na pasta **error-pages**, vamos criar um componente digitando no terminal o comando abaixo do **Angular CLI**:

```
ng g component error-pages/internal-server --skip-tests
```

Então, vamos modificar o arquivo `\src\app\error-pages\internal-server\internal-server.component.ts`:

```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-internal-server',
  templateUrl: './internal-server.component.html',
  styleUrls: ['./internal-server.component.css']
})
export class InternalServerComponent implements OnInit {
  errorMessage: string = "500 SERVER ERROR, CONTACT ADMINISTRATOR!!!!";

  constructor() { }

  ngOnInit(): void {
  }
}
```

Agora, vamos modificar o arquivo `internal-server.component.html`:

```
<p> {{errorMessage}} </p>
```

Além disso, vamos modificar o arquivo `internal-server.component.css`:

```
p {
  font-weight: bold;
  font-size: 50px;
  text-align: center;
  color: #c72d2d;
}
```

Finalmente, vamos adicionar um **import** e modificar o vetor **routes** no arquivo **\src\app\app-routing.module.ts**:

```
import { HomeComponent } from './home/home.component';

import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { NotFoundComponent } from './error-pages/not-found/not-found.component';
import { InternalServerErrorComponent } from './error-pages/internal-server/internal-server.component';

const routes: Routes = [
  { path: 'home', component: HomeComponent },
  { path: 'owner', loadChildren:()=>import('./owner/owner.module').then(m=>m.OwnerModule) },
  { path: '404', component: NotFoundComponent },
  { path: '500', component: InternalServerErrorComponent },
  { path: '', redirectTo: '/home', pathMatch: 'full' },
  { path: '**', redirectTo: '/404', pathMatch: 'full' }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Criamos nosso componente e agora é hora de criar um serviço para tratamento de erros.

Criando um Serviço Angular para Tratamento de Erros

Na pasta **shared/services**, vamos criar um serviço e nomeá-lo **error-handler**:

```
ng g service shared/services/error-handler --skip-tests
```

Em seguida, vamos modificar o arquivo **\src\app\shared\services\error-handler.service.ts**, que acabamos de criar:

```
import { HttpResponse } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { Router } from '@angular/router';

@Injectable({
  providedIn: 'root'
})
export class ErrorHandlerService {
  public errorMessage: string = '';

  constructor(private router: Router) { }
```

```

public handleError = (error: HttpResponse) => {
  if (error.status === 500) {
    this.handle500Error(error);
  }
  else if (error.status === 404) {
    this.handle404Error(error)
  }
  else {
    this.handleOtherError(error);
  }
}

private handle500Error = (error: HttpResponse) => {
  this.createErrorMessage(error);
  this.router.navigate(['/500']);
}

private handle404Error = (error: HttpResponse) => {
  this.createErrorMessage(error);
  this.router.navigate(['/404']);
}

private handleOtherError = (error: HttpResponse) => {
  this.createErrorMessage(error); //TODO: this will be fixed later;
}

private createErrorMessage = (error: HttpResponse) => {
  this.errorMessage = error.error ? error.error : error.statusText;
}
}

```

Primeiro de tudo, injetamos o **Router**, que usamos para redirecionar o usuário para outras páginas do código. Na função **handleError()**, verificamos o código de status do erro e, com base nisso, chamamos o método privado correto para lidar com esse erro. As funções **handle404Error()** e **handle500Error()** são responsáveis por preencher a propriedade **errorMessage**. Vamos usar essa propriedade como uma mensagem de erro modal ou página de erro. Vamos tratar da função **handleOtherError()** mais tarde, daí o comentário no código.

Se você se lembrar do arquivo **owner-list.component.ts**, estamos buscando todos os **owners** no servidor. Mas não há tratamento de erros nesse arquivo. Então, vamos continuar modificando o arquivo **owner-list.component.ts** para implementar a funcionalidade de tratamento de erros do Angular.

```

import { Component, OnInit } from '@angular/core';

import { Owner } from './../../_interfaces/owner.model';
import { OwnerRepositoryService } from './../../shared/services/owner-repository.service';
import { ErrorHandlerService } from './../../shared/services/error-handler.service';
import { HttpResponse } from '@angular/common/http';

@Component({
  selector: 'app-owner-list',
  templateUrl: './owner-list.component.html',
  styleUrls: ['./owner-list.component.css']
})

```

```

export class OwnerListComponent implements OnInit {
  owners: Owner[];
  errorMessage: string = '';

  constructor(private repository: OwnerRepositoryService, private errorHandler:
ErrorHandlerService) { }

  ngOnInit(): void {
    this.getAllOwners();
  }

  private getAllOwners = () => {
    const apiAddress: string = 'api/owner';
    this.repository.getOwners(apiAddress)
      .subscribe({
        next: (own: Owner[]) => this.owners = own,
        error: (err: HttpResponse) => {
          this.errorHandler.handleError(err);
          this.errorMessage = this.errorHandler.errorMessage;
        }
      })
  }
}

```

É isso. Temos que prestar atenção agora, estamos passando um objeto **JSON** dentro da função **subscribe**. A propriedade **next** será acionada se a resposta for bem-sucedida e a propriedade **error** manipulará a resposta de erro.

Podemos experimentar alterar o código no método do servidor **GetAllOwners**. Apenas a primeira linha de código, podemos adicionar **return NotFound()** ou **return StatusCode(500, "Some message")**, e vamos ser redirecionados para a página de erro correta.

Preparação para o Componente Owner-Details

Vamos continuar criando o componente **owner-details**:

```

ng g component owner/owner-details --skip-tests

```

Para habilitar o roteamento para esse componente, precisamos modificar o arquivo **\src\app\owner\owner-routing.module.ts**:

```

import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { OwnerDetailsComponent } from '../owner-details/owner-details.component';

import { OwnerListComponent } from '../owner-list/owner-list.component';

const routes: Routes = [
  { path: 'list', component: OwnerListComponent },
  { path: 'details/:id', component: OwnerDetailsComponent }
];

```

```
@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class OwnerRoutingModule { }
```

Como você pode ver, o novo caminho tem o parâmetro `id`. Então, quando clicamos no botão **Details**, vamos passar isso para a nossa rota e buscar o **owner** com esse `id` no componente **OwnerDetails**.

Mas, para poder fazer isso, precisamos adicionar uma nova interface à pasta `_interfaces`. Adicione na pasta `\src\app_interfaces` um arquivo com o nome **account.model.ts** e faça a inclusão do código abaixo:

```
export interface Account{
  id: string;
  dateCreated: Date;
  accountType: string;
  ownerId?: string;
}
```

Agora vamos modificar a interface **Owner** no arquivo `\src\app_interfaces`:

```
import { Account } from './account.model';
export interface Owner {
  id: string;
  name: string;
  dateOfBirth: Date;
  address: string;

  accounts?: Account[];
}
```

Ao usar um ponto de interrogação, estamos tornando nossa propriedade opcional.

Para continuar, vamos alterar o arquivo `\src\app\owner\owner-list\owner-list.component.html`, alterando o código do botão **Details**:

```
<td><button type="button" id="details" class="btn btn-primary"
  (click)="getOwnerDetails(owner.id)">Details</button></td>
```

Em um evento de clique, chamamos a função **getOwnerDetails(owner.id)** e passamos o `id` do **owner** como parâmetro. Portanto, precisamos lidar com esse evento de clique em nosso arquivo **owner-list.component.ts**.

Abra o arquivo `\src\app\owner\owner-list\owner-list.component.ts` e faça as modificações em destaque abaixo, incluindo o **import { Router }**, a injeção de construtor e a função **getOwnerDetails(id)**:

```

import { Component, OnInit } from '@angular/core';

import { Owner } from './../../../_interfaces/owner.model';
import { OwnerRepositoryService } from './../../../shared/services/owner-repository.service';
import { ErrorHandlerService } from './../../../shared/services/error-handler.service';
import { HttpResponse } from '@angular/common/http';
import { Router } from '@angular/router';

@Component({
  selector: 'app-owner-list',
  templateUrl: './owner-list.component.html',
  styleUrls: ['./owner-list.component.css']
})
export class OwnerListComponent implements OnInit {
  owners: Owner[];
  errorMessage: string = '';

  constructor(private repository: OwnerRepositoryService,
               private errorHandler: ErrorHandlerService,
               private router: Router) { }

  ngOnInit(): void {
    this.getAllOwners();
  }

  private getAllOwners = () => {
    const apiAddress: string = 'api/owner';
    this.repository.getOwners(apiAddress)
      .subscribe({
        next: (own: Owner[]) => this.owners = own,
        error: (err: HttpResponse) => {
          this.errorHandler.handleError(err);
          this.errorMessage = this.errorHandler.errorMessage;
        }
      })
  }

  public getOwnerDetails = (id) => {
    const detailsUrl: string = `/owner/details/${id}`;
    this.router.navigate([detailsUrl]);
  }
}

```

Criamos um **URI** para nosso componente de detalhes com o parâmetro **id** e, em seguida, chamamos a função de navegação para navegar até esse componente.

Finalmente, vamos apenas adicionar mais uma função para buscar um único **owner** dentro do serviço de repositório no arquivo `src\app\shared\services\owner-repository.service.ts`.

Adicione a função **getOwner()** do código abaixo, dentro da classe **OwnerRepositoryService**, do arquivo mencionado acima logo depois da função **getOwners()**:


```

public getOwner = (route: string) => {
  return this.http.get<Owner>(this.createCompleteRoute(route, this.envUrl.urlAddress));
}

```

Implementação do Componente Owner-Details

Temos todo o código para suportar o componente **owner-details**. Agora é hora de implementar a lógica de negócios dentro desse componente.

Em primeiro lugar, vamos modificar o arquivo `\src\app\owner\owner-details\owner-details.component.ts`:

```

import { HttpResponse } from '@angular/common/http';
import { Component, OnInit } from '@angular/core';
import { Owner } from './../../_interfaces/owner.model';
import { Router, ActivatedRoute } from '@angular/router';
import { OwnerRepositoryService } from './../../shared/services/owner-repository.service';
import { ErrorHandlerService } from './../../shared/services/error-handler.service';

@Component({
  selector: 'app-owner-details',
  templateUrl: './owner-details.component.html',
  styleUrls: ['./owner-details.component.css']
})
export class OwnerDetailsComponent implements OnInit {
  owner: Owner;
  errorMessage: string = '';

  constructor(private repository: OwnerRepositoryService, private router: Router,
    private activeRoute: ActivatedRoute, private errorHandler: ErrorHandlerService) { }

  ngOnInit() {
    this.getOwnerDetails()
  }

  getOwnerDetails = () => {
    const id: string = this.activeRoute.snapshot.params['id'];
    const apiUrl: string = `api/owner/${id}/account`;

    this.repository.getOwner(apiUrl)
      .subscribe({
        next: (own: Owner) => this.owner = own,
        error: (err: HttpResponse) => {
          this.errorHandler.handleError(err);
          this.errorMessage = this.errorHandler.errorMessage;
        }
      })
  }
}

```

É praticamente a mesma lógica que no arquivo **owner-list.component.ts**, exceto que agora temos o **ActivatedRoute** importado porque temos que obter nosso **id** da rota.

Depois de executarmos a função **getOwnerDetails**, vamos armazenar o objeto com todas as contas relacionadas dentro da propriedade **owner**.

Tudo o que temos a fazer é modificar o arquivo **\src\app\owner\owner-details\owner-details.component.html**:

```
<div class="card card-body bg-light mb-2 mt-2">
  <div class="row">
    <div class="col-md-3">
      <strong>Owner name:</strong>
    </div>
    <div class="col-md-3">
      {{owner?.name}}
    </div>
  </div>
  <div class="row">
    <div class="col-md-3">
      <strong>Date of birth:</strong>
    </div>
    <div class="col-md-3">
      {{owner?.dateOfBirth | date: 'dd/MM/yyyy'}}
    </div>
  </div>
  <div class="row" *ngIf='owner?.accounts.length <= 2; else advancedUser'>
    <div class="col-md-3">
      <strong>Type of user:</strong>
    </div>
    <div class="col-md-3">
      <span class="text-success">Beginner user.</span>
    </div>
  </div>
  <ng-template #advancedUser>
    <div class="row">
      <div class="col-md-3">
        <strong>Type of user:</strong>
      </div>
      <div class="col-md-3">
        <span class="text-info">Advanced user.</span>
      </div>
    </div>
  </ng-template>
</div>

<div class="row">
  <div class="col-md-12">
    <div class="table-responsive">
      <table class="table table-striped">
        <thead>
          <tr>
            <th>Account type</th>
```

```

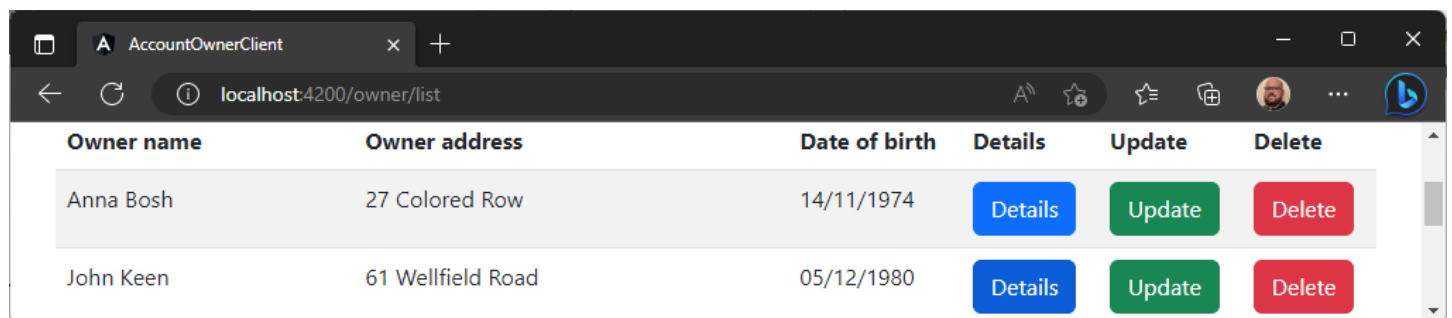
        <th>Date created</th>
      </tr>
    </thead>
    <tbody>
      <tr *ngFor="let account of owner?.accounts">
        <td>{{account?.accountType}}</td>
        <td>{{account?.dateCreated | date: 'dd/MM/yyyy'}}</td>
      </tr>
    </tbody>
  </table>
</div>
</div>
</div>

```

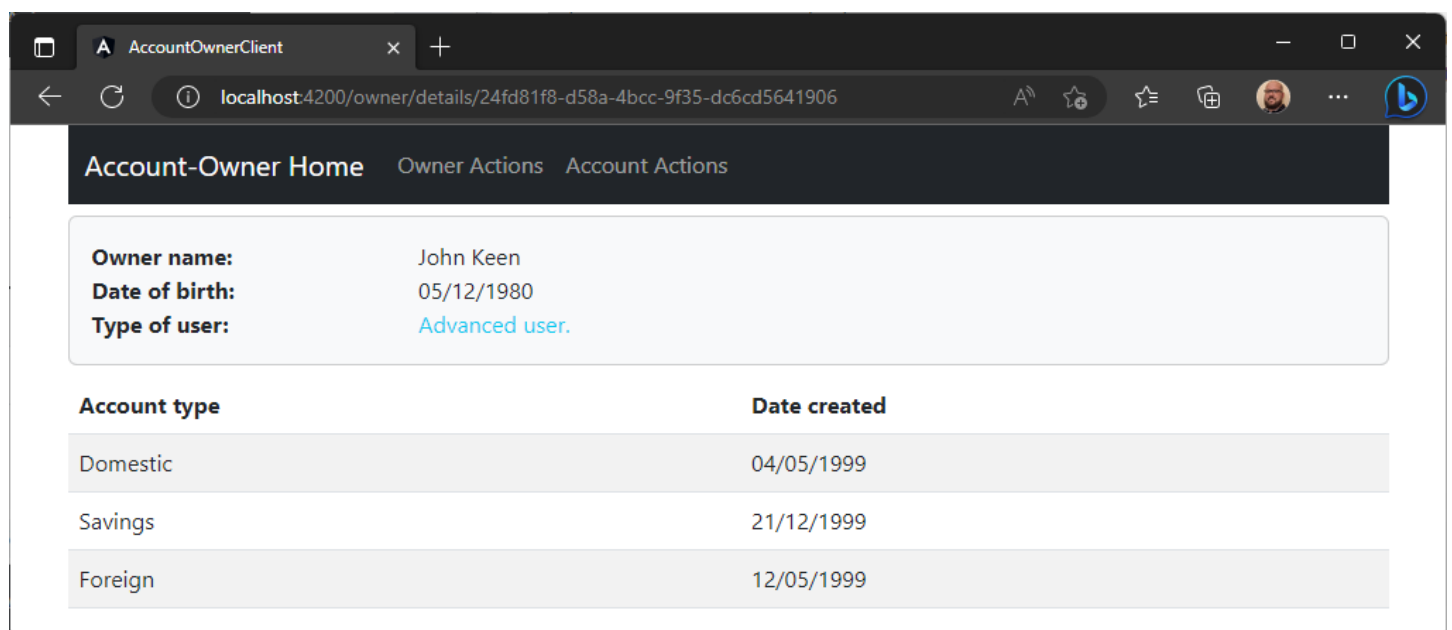
Aqui, exibimos a entidade **owner** com todos os dados necessários. Além disso, se o **owner** tiver mais de duas contas, mostraremos condicionalmente um modelo diferente (**#advancedUser**) para esse campo. Finalmente, exibimos todas as contas relacionadas a este **owner**:

Agora, salve todos os arquivos e no terminal (**lembrando de deixar o backend rodando em outro terminal**) execute o comando de execução da aplicação, conferindo se o caminho apresentado é da pasta **frontend\AccountOwnerClient**, para testes:

```
ng serve -o
```



Owner name	Owner address	Date of birth	Details	Update	Delete
Anna Bosh	27 Colored Row	14/11/1974	Details	Update	Delete
John Keen	61 Wellfield Road	05/12/1980	Details	Update	Delete



Account-Owner Home Owner Actions Account Actions

Owner name: John Keen

Date of birth: 05/12/1980

Type of user: [Advanced user.](#)

Account type	Date created
Domestic	04/05/1999
Savings	21/12/1999
Foreign	12/05/1999

Decoradores e Diretiva de Entrada e Saída do Angular

Os decoradores angulares de entrada e saída são muito importantes ao estabelecer uma relação entre os componentes pai e filho em nossas aplicações.

Ao desenvolver nosso projeto, às vezes nossos componentes podem se tornar grandes e difíceis de ler. Então, é sempre uma boa escolha dividir esse componente grande em alguns menores. Além disso, componentes menores podem ser reutilizados em outros componentes, portanto, criar a relação pai-filho é uma ideia muito boa. O componente filho depende do componente pai e, por isso, eles fazem uma parte coerente.

Criar componentes filhos usando decoradores de entrada e saída do Angular será nosso objetivo neste parte.

Visão Geral

Nas situações em que queremos enviar algum conteúdo de um componente pai para um componente filho, precisamos usar o decorador **@Input** em um componente filho para fornecer uma ligação de propriedade entre esses componentes. Além disso, podemos ter alguns eventos em um componente filho que refletem seu comportamento de volta a um componente pai. Para isso, vamos usar o decorador **@Output** com o **EventEmitter**.

Para lidar com mensagens de sucesso e mensagens de erro (além de 500 ou 404), vamos criar uma janela modal com componentes filhos. Vamos reutilizá-los em todos os componentes que exijam a exibição desses tipos de mensagens. Quando queremos registrar nosso componente reutilizável, é uma boa prática criar um módulo compartilhado e registrar e exportar nossos componentes dentro desse módulo. Então, podemos usar esses componentes reutilizáveis em qualquer componente de nível superior que quisermos, registrando o módulo compartilhado dentro de um módulo responsável por esse componente de ordem superior.

Dividindo o OwnerDetails com Decoradores de Entrada e Saída do Angular

Atualmente, temos nosso componente **OwnerDetails** que mostra os dados do **owner** e **accounts** na página. Podemos dividir isso, tornando nosso componente mais limpo e fácil de manter.

Para fazer isso, vamos começar com a criação de um novo componente:

```
ng g component owner/owner-details/owner-accounts --skip-tests
```

Colocamos os arquivos desse componente dentro da pasta **owner-details** porque só vamos usá-lo para extrair conteúdo do componente **OwnerDetails**.

Agora, vamos modificar o arquivo **owner-accounts.component.ts**, conforme o código abaixo:

```
import { Component, Input, OnInit } from '@angular/core';

import { Account } from 'src/app/_interfaces/account.model';

@Component({
  selector: 'app-owner-accounts',
  templateUrl: './owner-accounts.component.html',
  styleUrls: ['./owner-accounts.component.css']
})
```

```
export class OwnerAccountsComponent implements OnInit {
  @Input() accounts: Account[];

  constructor(){ }

  ngOnInit(): void {
  }
}
```

Adicionamos uma única propriedade do tipo vetor de **Account** e a decoramos com o decorador **@Input**. Em seguida, vamos recortar a tabela que mostra as **accounts** do arquivo **owner-details.component.html** e substituí-la pelo seletor do componente **OwnerAccountsComponent**, criado acima. Recorte o código em destaque abaixo e substitua pela linha do próximo trecho de código:

```
<ng-template #advancedUser>
  <div class="row">
    <div class="col-md-3">
      <strong>Type of user:</strong>
    </div>
    <div class="col-md-3">
      <span class="text-info">Advanced user.</span>
    </div>
  </div>
</ng-template>
</div>

<div class="row">
  <div class="col-md-12">
    <div class="table-responsive">
      <table class="table table-striped">
        <thead>
          <tr>
            <th>Account type</th>
            <th>Date created</th>
          </tr>
        </thead>
        <tbody>
          <tr *ngFor="let account of owner?.accounts">
            <td>{{account?.accountType}}</td>
            <td>{{account?.dateCreated | date: 'dd/MM/yyyy'}}</td>
          </tr>
        </tbody>
      </table>
    </div>
  </div>
</div>
```

Seletor do **OwnerAccountsComponent** a substituir a tabela:

```
<app-owner-accounts [accounts]="owner?.accounts"></app-owner-accounts>
```

Aqui, usamos o seletor **app-owner-accounts** para injetar o componente filho no componente pai. Além disso, usamos uma propriedade de associação para passar todas as **accounts** do objeto **owner** para a propriedade **@Input accounts** no componente filho.

Depois disso, podemos colar o código que cortamos do componente pai dentro do arquivo **owner-accounts.component.html**:

```
<div class="row">
  <div class="col-md-12">
    <div class="table-responsive">
      <table class="table table-striped">
        <thead>
          <tr>
            <th>Account type</th>
            <th>Date created</th>
          </tr>
        </thead>
        <tbody>
          <tr *ngFor="let account of accounts">
            <td>{{account?.accountType}}</td>
            <td>{{account?.dateCreated | date: 'dd/MM/yyyy'}}</td>
          </tr>
        </tbody>
      </table>
    </div>
  </div>
</div>
```

Também podemos notar que **não estamos mais usando o owner?.accounts** dentro da diretiva ***ngFor**, mas apenas a propriedade **accounts**.

Com isso em vigor, podemos iniciar nosso aplicativo e navegar até o componente **OwnerDetails**, e veremos **o mesmo resultado de antes**. Desta vez, dividimos a lógica em dois componentes.

Incorporar eventos do componente filho usando @Output decorador

Vamos usar um exemplo mais simples para mostrar como podemos emitir eventos do componente filho para o componente pai.

Para começar, vamos adicionar algumas modificações dentro do arquivo **owner-accounts.component.ts**:

```
import { Component, EventEmitter, Input, OnInit, Output } from '@angular/core';

import { Account } from 'src/app/_interfaces/account.model';

@Component({
  selector: 'app-owner-accounts',
  templateUrl: './owner-accounts.component.html',
  styleUrls: ['./owner-accounts.component.css']
})
```

```
export class OwnerAccountsComponent implements OnInit {
  @Input() accounts: Account[];
  @Output() onAccountClick: EventEmitter<Account> = new EventEmitter();

  constructor(){ }

  ngOnInit(): void {
  }

  onAccountClicked = (account: Account) => {
    this.onAccountClick.emit(account);
  }
}
```

Usamos o decorador **@Output** com o **EventEmitter** para emitir um evento do componente filho para o componente pai. Além disso, fornecemos um tipo para o **EventEmitter**, afirmamos que ele pode emitir apenas o tipo de dados **Account**. Além disso, criamos uma função **onAccountClicked**, onde aceitamos uma **account** e chamamos a função **emit** para emitir esse evento para o componente pai.

Para poder chamar a função **onAccountClicked**, temos que modificar o arquivo **owner-accounts.component.html**:

```
<div class="row">
  <div class="col-md-12">
    <div class="table-responsive">
      <table class="table table-striped">
        <thead>
          <tr>
            <th>Account type</th>
            <th>Date created</th>
          </tr>
        </thead>
        <tbody>
          <tr *ngFor="let account of accounts" (click)="onAccountClicked(account)">
            <td>{{account?.accountType}}</td>
            <td>{{account?.dateCreated | date: 'dd/MM/yyyy'}}</td>
          </tr>
        </tbody>
      </table>
    </div>
  </div>
</div>
```

Apenas adicionamos o evento **click** à linha dentro da tabela.

Finalmente, para ter um cursor de ponteiro quando passamos o mouse sobre as linhas da nossa conta, vamos modificar o arquivo **owner-accounts.component.css**:

```
tbody tr:hover {
  cursor: pointer;
}
```

Para que nosso emissor funcione, temos que assiná-lo a partir do nosso componente pai.

A primeira coisa que vamos fazer é adicionar uma única função (conforme o código em destaque abaixo) dentro do arquivo **owner-details.component.ts**:

```
import { HttpResponseResponse } from '@angular/common/http';
import { Component, OnInit } from '@angular/core';
import { Owner } from '../../../_interfaces/owner.model';
import { Router, ActivatedRoute } from '@angular/router';
import { OwnerRepositoryService } from '../../../shared/services/owner-repository.service';
import { ErrorHandlerService } from '../../../shared/services/error-handler.service';
import { Account } from 'src/app/_interfaces/account.model';

@Component({
  selector: 'app-owner-details',
  templateUrl: './owner-details.component.html',
  styleUrls: ['./owner-details.component.css']
})
export class OwnerDetailsComponent implements OnInit {
  owner: Owner;
  errorMessage: string = '';

  constructor(private repository: OwnerRepositoryService, private router: Router,
    private activeRoute: ActivatedRoute, private errorHandler: ErrorHandlerService) { }

  ngOnInit() {
    this.getOwnerDetails()
  }

  getOwnerDetails = () => {
    const id: string = this.activeRoute.snapshot.params['id'];
    const apiUrl: string = `api/owner/${id}/account`;

    this.repository.getOwner(apiUrl)
      .subscribe({
        next: (own: Owner) => this.owner = own,
        error: (err: HttpResponseResponse) => {
          this.errorHandler.handleError(err);
          this.errorMessage = this.errorHandler.errorMessage;
        }
      })
  }

  printToConsole= (param: Account) => {
    console.log('Account parameter from the child component', param)
  }
}
```

E então, vamos apenas ligar os pontos no arquivo **owner-details.component.html**:


```

<div class="card card-body bg-light mb-2 mt-2">
  <div class="row">
    <div class="col-md-3">
      <strong>Owner name:</strong>
    </div>
    <div class="col-md-3">
      {{owner?.name}}
    </div>
  </div>
  <div class="row">
    <div class="col-md-3">
      <strong>Date of birth:</strong>
    </div>
    <div class="col-md-3">
      {{owner?.dateOfBirth | date: 'dd/MM/yyyy'}}
    </div>
  </div>
  <div class="row" *ngIf='owner?.accounts.length <= 2; else advancedUser'>
    <div class="col-md-3">
      <strong>Type of user:</strong>
    </div>
    <div class="col-md-3">
      <span class="text-success">Beginner user.</span>
    </div>
  </div>
  <ng-template #advancedUser>
    <div class="row">
      <div class="col-md-3">
        <strong>Type of user:</strong>
      </div>
      <div class="col-md-3">
        <span class="text-info">Advanced user.</span>
      </div>
    </div>
  </ng-template>
</div>

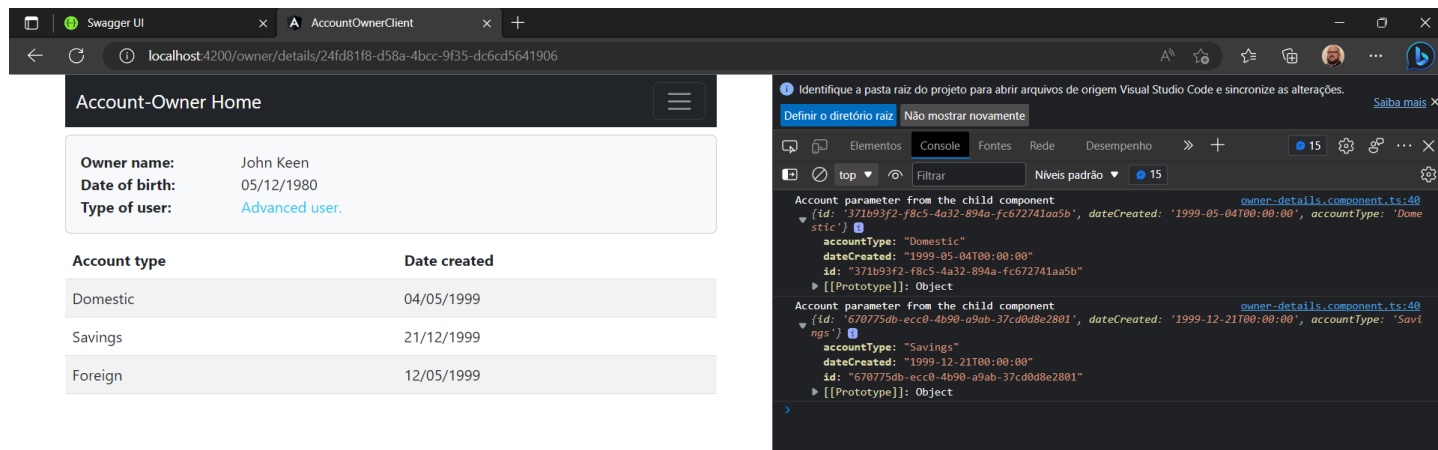
<app-owner-accounts [accounts]="owner?.accounts"
  (onAccountClick)="printToConsole($event)"></app-owner-accounts>

```

Aqui, criamos o evento **com o mesmo nome de nossa propriedade @Output no componente filho** e atribuímos uma função local **printToConsole** a ele. Além disso, usamos **\$event** como parâmetro para aceitar o objeto emitido do componente filho.

Neste ponto, podemos iniciar nosso aplicativo e, uma vez que navegamos para o componente **OwnerDetails** (clicando no botão **Details** de um dos **owners**), podemos ver que assim que clicarmos em qualquer **account**, registraremos uma mensagem no console.

Conforme explicado acima, executando o **backend** e o **frontend**, podemos clicar no link **Owner Actions** e em seguida, no botão **Details** do **Owner John Keen**. Assim que a página de detalhes é carregada, podemos notar que ao passar o mouse sobre as **Accounts**, o ponteiro muda para sua forma **“clícavel”**, clicando nas contas, no **console** das **“ferramentas do desenvolvedor”** os dados daquela **account** são carregados.



Criando o módulo compartilhado

Como dissemos, para os componentes compartilhados, é sempre uma boa prática criar um módulo compartilhado. Então, vamos começar criando esse módulo compartilhado na pasta compartilhada:

```
ng g module shared --module owner
```

Este comando irá criar o arquivo `\src\app\shared\shared.module.ts` e incluir uma importação deste como **SharedModule** dentro do arquivo `\src\app\owner\owner.module.ts`.

Então vamos modificar o arquivo **shared.module.ts**:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

@NgModule({
  declarations: [],
  imports: [
    CommonModule
  ],
  exports: []
})
export class SharedModule { }
```

Componente modal de erro

Vamos executar o comando do **AngularCLI** para criar um componente modal de erro:

```
ng g component shared/modals/error-modal --skip-tests
```

Além de criar arquivos de componentes, esse comando importará um novo componente para o módulo compartilhado (**shared.module.ts**). Mas precisamos exportá-lo manualmente, para isso abra o arquivo **shared.module.ts** e faça as alterações em destaque abaixo:

```

import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { ModalModule } from 'ngx-bootstrap/modal';
import { ErrorModalComponent } from './modals/error-modal/error-modal.component';

@NgModule({
  declarations: [
    ErrorModalComponent
  ],
  imports: [
    CommonModule,
    ModalModule.forRoot()
  ],
  exports: [
    ErrorModalComponent
  ]
})
export class SharedModule { }

```

No vetor **exports**, exportamos nosso componente. Além disso, como vamos usar o componente **modal do ngx-bootstrap**, nós o injetamos dentro do vetor **imports** usando a função **forRoot()**.

Agora, vamos modificar o arquivo `\src\app\shared\modals\error-modal\error-modal.component.ts`:

```

import { Component, OnInit } from '@angular/core';
import { BsModalRef } from 'ngx-bootstrap/modal';

@Component({
  selector: 'app-error-modal',
  templateUrl: './error-modal.component.html',
  styleUrls: ['./error-modal.component.css']
})
export class ErrorModalComponent implements OnInit {
  modalHeaderText: string;
  modalBodyText: string;
  okButtonText: string;

  constructor(public bsModalRef: BsModalRef) { }

  ngOnInit(): void {
  }
}

```

Aqui nós temos três propriedades que iremos utilizar na **HTML** deste componente. Também injetamos a classe **BsModalRef** dentro do construtor, que também será utilizado no arquivo **HTML**, desta forma deixando-o público.

Vamos continuar modificando o arquivo **error.modal.componet.html**:

```

<div class="modal-header">
  <h4 class="modal-title pull-left">{{modalHeaderText}}</h4>
  <button type="button" class="btn-close close pull-right" aria-label="Close"
(click)="bsModalRef.hide()">
    <span aria-hidden="true" class="visually-hidden">&times;</span>
  </button>
</div>
<div class="modal-body">
  {{modalBodyText}}
</div>
<div class="modal-footer">
  <button type="button" class="btn btn-danger"
(click)="bsModalRef.hide()">{{okButtonText}}</button>
</div>

```

Apenas um arquivo **HTML** simples que usa interpolação de cadeia de caracteres para adicionar o texto para o cabeçalho, corpo e botão. Além disso, podemos ver como usamos o **bsModalRef**, que injetamos dentro do construtor, para chamar a função **hide**, para ocultar o modal.

Criando o componente modal de sucesso

Para continuar, vamos criar um modal de sucesso da mesma forma que fizemos com o modal de erro:

```
ng g component shared/modals/success-modal --skip-tests
```

Temos que exportar o componente modal de sucesso no arquivo **shared.module.ts**:

```

import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { ModalModule } from 'ngx-bootstrap/modal';
import { ErrorModalComponent } from './modals/error-modal/error-modal.component';
import { SuccessModalComponent } from './modals/success-modal/success-modal.component';

@NgModule({
  declarations: [
    ErrorModalComponent,
    SuccessModalComponent
  ],
  imports: [
    CommonModule,
    ModalModule.forRoot()
  ],
  exports: [
    ErrorModalComponent,
    SuccessModalComponent
  ]
})
export class SharedModule { }

```

Então, vamos modificar nosso arquivo **success-modal.component.ts**:

```
import { BsModalRef } from 'ngx-bootstrap/modal';
import { Component, EventEmitter, OnInit } from '@angular/core';

@Component({
  selector: 'app-success-modal',
  templateUrl: './success-modal.component.html',
  styleUrls: ['./success-modal.component.css']
})
export class SuccessModalComponent implements OnInit {
  modalHeaderText: string;
  modalBodyText: string;
  okButtonText: string;
  redirectOnOk: EventEmitter<any> = new EventEmitter();

  constructor(private bsModalRef: BsModalRef) { }

  ngOnInit(): void {
  }

  onOkClicked = () => {
    this.redirectOnOk.emit();
    this.bsModalRef.hide();
  }
}
```

Vamos usar o componente de sucesso assim que executarmos as ações de criação, atualização ou exclusão com êxito e, pressionando o botão **OK**, redirecionaremos o usuário para o componente **owner-list**. É por isso que usamos a classe **EventEmitter**. Além disso, desta vez usamos o **bsModalRef** privado, porque o usamos apenas dentro do arquivo **.ts**.

Finalmente, vamos modificar o arquivo **success-modal.component.html**:

```
<div class="modal-header">
  <h4 class="modal-title pull-left">{{modalHeaderText}}</h4>
  <button type="button" class="btn-close close pull-right" aria-label="Close"
(click)="onOkClicked()">
    <span aria-hidden="true" class="visually-hidden">&times;</span>
  </button>
</div>
<div class="modal-body">
  {{modalBodyText}}
</div>
<div class="modal-footer">
  <button type="button" class="btn btn-success"
(click)="onOkClicked()">{{okButtonText}}</button>
</div>
```

Vamos usar esses dois componentes modais na próxima parte deste tutorial.

Diretivas

Mostraremos agora como usar diretivas para adicionar comportamento adicional a elementos em nosso aplicativo Angular. Então, vamos criar a diretiva **Append** na pasta compartilhada:

```
ng g directive shared/directives/append --skip-tests
```

Isso criará um arquivo com o decorador **@Directive** e o seletor **[appAppend]**. Vamos usar esse seletor para adicionar o comportamento adicional aos elementos dentro do componente **OwnerDetails**.

Abra o arquivo `\src\app\owner\owner-details\owner-details.component.html` e faça as alterações em destaque:

```
<div class="card card-body bg-light mb-2 mt-2">
  <div class="row">
    <div class="col-md-3">
      <strong>Owner name:</strong>
    </div>
    <div class="col-md-3">
      {{owner?.name}}
    </div>
  </div>
  <div class="row">
    <div class="col-md-3">
      <strong>Date of birth:</strong>
    </div>
    <div class="col-md-3">
      {{owner?.dateOfBirth | date: 'dd/MM/yyyy'}}
    </div>
  </div>
  <div class="row" *ngIf='owner?.accounts.length <= 2; else advancedUser'>
    <div class="col-md-3">
      <strong>Type of user:</strong>
    </div>
    <div class="col-md-3">
      <span [appAppend]="owner" class="text-success">Beginner user.</span>
    </div>
  </div>
  <ng-template #advancedUser>
    <div class="row">
      <div class="col-md-3">
        <strong>Type of user:</strong>
      </div>
      <div class="col-md-3">
        <span [appAppend]="owner" class="text-info">Advanced user.</span>
      </div>
    </div>
  </ng-template>
</div>

<app-owner-accounts [accounts]="owner?.accounts"
(onAccountClick)="printToConsole($event)"></app-owner-accounts>
```

Como você pode ver, adicionamos o seletor **[appAppend]** dentro de ambos os elementos **span** e, passamos o objeto **owner** como um parâmetro para a nossa diretiva.

Agora, temos que modificar o arquivo de diretiva. Abra e modifique o código do arquivo `\src\app\shared\directives\append.directive.ts`, de acordo com o código abaixo:

```
import { Directive, ElementRef, Input, OnChanges, Renderer2, SimpleChanges } from
'@angular/core';
import { Owner } from '../_interfaces/owner.model';

@Directive({
  selector: '[appAppend]'
})
export class AppendDirective implements OnChanges {
  @Input('appAppend') ownerParam: Owner;

  constructor(private element: ElementRef, private renderer: Renderer2) { }

  ngOnChanges(changes: SimpleChanges) {
    if(changes.ownerParam.currentValue){
      const accNum = changes.ownerParam.currentValue.accounts.length;
      const span = this.renderer.createElement('span');
      const text = this.renderer.createText(` (${accNum}) accounts`);

      this.renderer.appendChild(span, text);
      this.renderer.appendChild(this.element.nativeElement, span);
    }
  }
}
```

Aqui temos a propriedade **ownerParam** que decoramos com o decorador **@Input**. Fazemos isso para aceitar o parâmetro do componente pai. Então, dentro do construtor, injetamos as classes **ElementRef** e **Renderer2**. Usamos a classe **ElementRef** para fazer referência a um elemento ao qual aplicamos nossa diretiva. Além disso, usamos o **Renderer2** para manipular os elementos sem tocar diretamente no **DOM**. Então, como você pode ver, não há necessidade de **JQuery** para manipular os elementos **DOM** – **Angular** nos fornece todas as ferramentas que precisamos.

Após a modificação do construtor, adicionamos um novo método de ciclo de vida **ngOnChanges**. Esse método é chamado antes do gancho do ciclo de vida **ngOnInit** e toda vez que uma ou mais propriedades de entrada são alteradas. Dentro dele, verificamos se nossa propriedade de entrada **currentValue** tem seu valor preenchido. Se isso acontecer, extraímos o número de **accounts**, e usamos **renderer** para criar elementos **span** e elementos **text** e os combinamos.

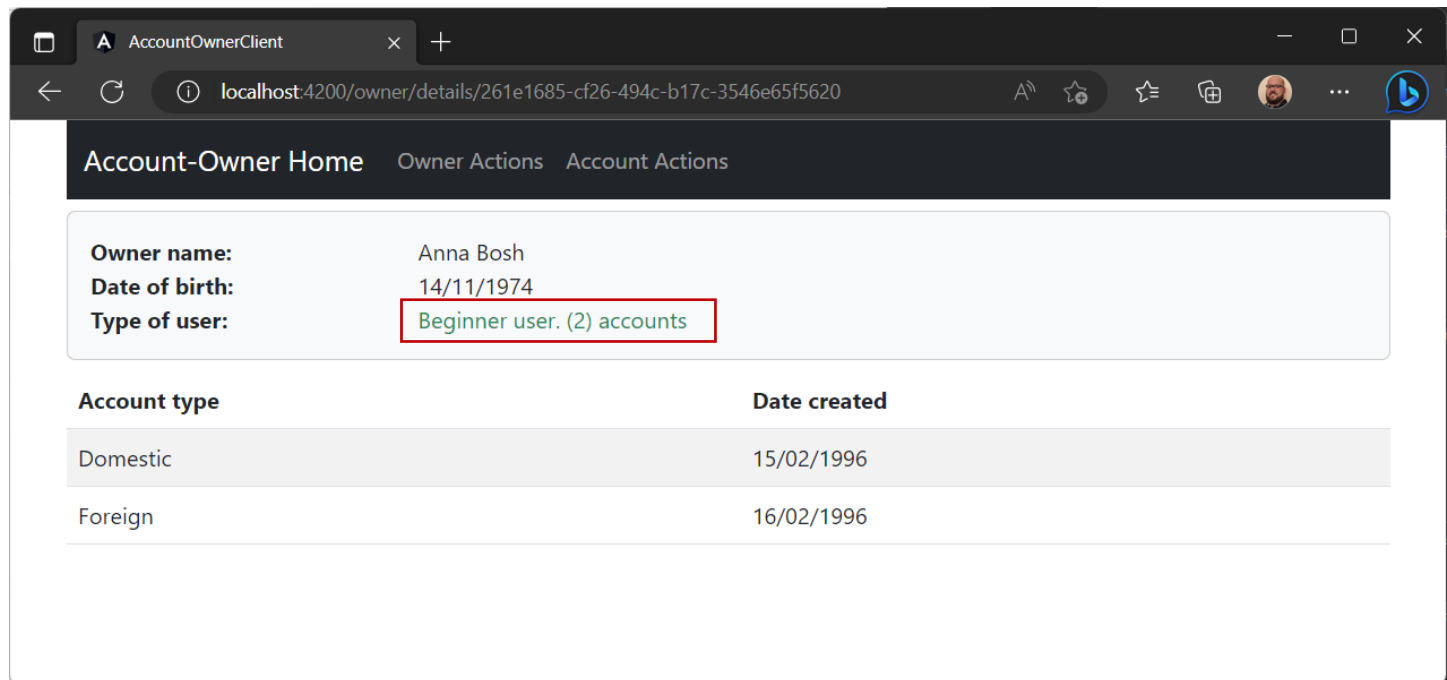
Finalmente, usamos a mesma propriedade **renderer** para combinar o nosso **nativeElement** com o elemento **span** recém-criado.

Agora precisamos incluir nossa diretiva nos módulos compartilhados, para isso abra o arquivo `\src\app\shared\shared.module.ts` e no vetor **exports** faça a inclusão do **AppendDirective**:

```
exports: [
  ErrorModalComponent,
```

```
SuccessModalComponent,  
AppendDirective  
]
```

Agora podemos iniciar nossos aplicativos e, uma vez que navegamos para a página de detalhes de qualquer usuário, veremos nossa diretiva em ação:



Veja que temos novas informações ao lado da descrição do usuário. Para usuários diferentes, obteremos números de conta diferentes.

Validação de Formulário no Angular e Solicitação POST

Preparação para o componente Create Owner

Vamos começar criando nosso componente dentro da pasta **owner**. Para fazer isso, execute o comando:

```
ng g component owner/owner-create --skip-tests
```

Em seguida, vamos modificar o módulo de roteamento do **owner** adicionando uma nova rota, através da alteração do arquivo **owner-routing.module.ts**:

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { OwnerDetailsComponent } from '../owner-details/owner-details.component';

import { OwnerListComponent } from '../owner-list/owner-list.component';
import { OwnerCreateComponent } from '../owner-create/owner-create.component';

const routes: Routes = [
  { path: 'list', component: OwnerListComponent },
  { path: 'details/:id', component: OwnerDetailsComponent },
  { path: 'create', component: OwnerCreateComponent }
];

@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class OwnerRoutingModule { }
```

Quando clicamos no link "**create**" dentro do arquivo **owner-list.component.html**, queremos que o aplicativo nos direcione para a página de criação.

Então, vamos modificar a tag **<a>** dentro do arquivo **owner-list.component.html**, alterando apenas no começo do arquivo a tag **<a>** existente conforme código abaixo:

```
<div class="row">
  <div class="offset-10 col-md-2 mt-2">
    <a [routerLink]="['/owner/create']">Create owner</a>
  </div>
```

Sobre a validação de formulário e o ReactiveFormsModule

Agora, podemos começar a escrever o código para criar nossa entidade e validar o formulário. Há dois tipos de validação no Angular: **template-driven validation** (validação orientada por modelo) e **reactive form**

validation (validação de formulário reativo). Em nosso projeto, vamos usar a validação de formulário reativo porque um arquivo **HTML** é mais fácil de ler. Além disso, ele não torna os arquivos **HTML** muito "sujos" com muitas linhas de código e toda a validação está no componente, o que facilita a manutenção.

Pouco antes de modificarmos nosso componente, precisamos modificar o arquivo **owner.module.ts**. Vamos importar o **ReactiveFormsModule** porque este é o módulo que suporta a validação de formulário reativo. Aproveitando a alteração do arquivo também vamos incluir o módulo de **DatePicker** do **ngx-bootstrap**:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

import { OwnerRoutingModule } from './owner-routing.module';
import { OwnerListComponent } from './owner-list/owner-list.component';
import { OwnerDetailsComponent } from './owner-details/owner-details.component';
import { OwnerAccountsComponent } from './owner-details/owner-accounts/owner-accounts.component';
import { SharedModule } from '../shared/shared.module';
import { OwnerCreateComponent } from './owner-create/owner-create.component';
import { ReactiveFormsModule } from '@angular/forms';
import { BsDatepickerModule } from 'ngx-bootstrap/datepicker';

@NgModule({
  declarations: [
    OwnerListComponent,
    OwnerDetailsComponent,
    OwnerAccountsComponent,
    OwnerCreateComponent
  ],
  imports: [
    CommonModule,
    OwnerRoutingModule,
    SharedModule,
    ReactiveFormsModule,
    BsDatepickerModule.forRoot()
  ]
})
export class OwnerModule { }
```

Além disso, temos que criar outra interface na pasta **src\app\interfaces**, crie um arquivo com o nome **ownerForCreation.model.ts** e faça as alterações abaixo:

```
export interface OwnerForCreation {
  name: string;
  dateOfBirth: string;
  address: string;
}
```

Validação de Formulário no Angular – HTML

Vamos continuar com a modificação do arquivo **owner-create.component.html**:

Agora vamos explicar esse código. No elemento **form**, criamos o **formGroup** com o nome **ownerForm**. Esse grupo de formulário contém todos os controles que precisamos validar em nosso formulário. Além disso, com o que (**ngSubmit**) chamamos de uma função quando o usuário pressionar o botão enviar. Como parâmetro para essa função, enviamos o valor do **ownerForm** que contém todos os controles com os dados necessários para a validação.

Há um atributo **formControlName** dentro de cada controle. Esse atributo representa o nome do controle que vamos validar dentro do **ownerForm** e é um atributo obrigatório. Além disso, nas tags ****, exibimos mensagens de erro, se houver.

Uma coisa a prestar atenção é o elemento de entrada DateOfBirth. Aqui usamos a diretiva **bsDatePicker** do **ngx-bootstrap**, para anexar o **BsDatepickerModule** a este componente de entrada. Para que isso funcione, incluímos anteriormente o módulo dentro do arquivo **owner.module.ts**.

Há muito mais funcionalidades que podemos usar com o seletor de data do **ngx-bootstrap** e, para saber mais sobre elas, você pode ler [esta documentação](#).

Sobre os Erros e Botões no Formulário

Os erros serão gravados na página somente se as funções **validateControl()** e **hasError()** retornarem **true** como resultado.

A função **validateControl()** vai verificar se o controle é inválido e a função **hasError()** vai verificar quais regras de validação estamos validando (obrigatório, comprimento máximo...). Ambas as funções são personalizadas, e vamos implementar no arquivo de componente (.ts). Há também um botão de envio que será desativado até que o formulário se torne válido e um botão de cancelamento que redirecionará o usuário para fora do formulário de criação.

Validação de Formulário no Angular (.ts) – Componente.

Agora, temos que implementar a lógica para todas as funções chamadas no arquivo de modelo. Vamos adicionar muita lógica ao arquivo **owner-create.component.ts**, então vamos tentar dividir a explicação em várias partes. Começemos pelas importações:

```
import { DatePipe } from '@angular/common';
import { HttpResponse } from '@angular/common/http';
import { Component, OnInit } from '@angular/core';
import { FormControl, FormGroup, Validators } from '@angular/forms';
import { Router } from '@angular/router';
import { BsModalRef, BsModalService, ModalOptions } from 'ngx-bootstrap/modal';
import { Owner } from 'src/app/_interfaces/owner.model';
import { OwnerForCreation } from 'src/app/_interfaces/ownerForCreation.model';
import { SuccessModalComponent } from 'src/app/shared/modals/success-modal/success-modal.component';
import { ErrorHandlerService } from 'src/app/shared/services/error-handler.service';
import { OwnerRepositoryService } from 'src/app/shared/services/owner-repository.service';
```

Muitas instruções de importação, em seguida veremos em todo o código porque usamos cada uma delas.

Agora, vamos inspecionar outra parte do arquivo:

```

export class OwnerCreateComponent implements OnInit {
  errorMessage: string = '';
  ownerForm: FormGroup;
  bsModalRef?: BsModalRef;

  constructor(private repository: OwnerRepositoryService, private errorHandler:
ErrorHandlerService,
    private router: Router, private datePipe: DatePipe, private modal: BsModalService) { }

  ngOnInit(): void {
    this.ownerForm = new FormGroup({
      name: new FormControl('', [Validators.required, Validators.maxLength(60)]),
      dateOfBirth: new FormControl('', [Validators.required]),
      address: new FormControl('', [Validators.required, Validators.maxLength(100)])
    });
  }
}

```

Assim que um componente é montado, estamos inicializando nossa variável nomeada **ownerForm** com todos os **FormControls**. Preste atenção que as chaves no objeto **ownerForm** são as mesmas que os nomes do atributo **formControlName** para todos os campos de entrada em um arquivo **.html**, o que é obrigatório. Além disso, eles têm o mesmo nome que as propriedades dentro do objeto de **owner** (**name**, **address** e **dataOfBirth**).

Ao instanciar um novo controle de formulário como um primeiro parâmetro, estamos fornecendo o valor do controle e, como segundo parâmetro, a matriz **Validators**, que contém todas as regras de validação para nossos controles.

Manipulando erros

Agora, vamos adicionar duas funções que chamamos em nosso arquivo de modelo para lidar com erros de validação, abaixo do **ngOnInit()**:

```

validateControl = (controlName: string) => {
  if (this.ownerForm.get(controlName).invalid && this.ownerForm.get(controlName).touched)
    return true;

  return false;
}

hasError = (controlName: string, errorName: string) => {
  if (this.ownerForm.get(controlName).hasError(errorName))
    return true;

  return false;
}

```

No método **validateControl()**, estamos verificando se o controle atual é inválido e tocado (não queremos mostrar um erro se o usuário não colocou o cursor dentro do controle). Além disso, a função **hasError()** verificará qual regra de validação o controle atual violou.

Processo de Criação, DatePipe e Invocação do Modal

Para continuar, vamos adicionar a função **createOwner** que chamamos no evento (**ngSubmit**), abaixo da função **hasError()**:

```
createOwner = (ownerFormValue) => {  
  if (this.ownerForm.valid)  
    this.executeOwnerCreation(ownerFormValue);  
}
```

Ele aceita o valor do formulário, verifica se o formulário é válido e, se ele é chama outra função privada passando o valor do formulário como um parâmetro.

Dito isso, vamos criar essa função privada, abaixo do **createOwner**:

```
private executeOwnerCreation = (ownerFormValue) => {  
  const owner: OwnerForCreation = {  
    name: ownerFormValue.name,  
    dateOfBirth: this.datePipe.transform(ownerFormValue.dateOfBirth, 'yyyy-MM-dd'),  
    address: ownerFormValue.address  
  }  
  const apiUrl = 'api/owner';  
  this.repository.createOwner(apiUrl, owner)  
    .subscribe({  
      next: (own: Owner) => {  
        const config: ModalOptions = {  
          initialState: {  
            modalHeaderText: 'Success Message',  
            modalBodyText: `Owner: ${own.name} created successfully`,  
            okButtonText: 'OK'  
          }  
        };  
        this.bsModalRef = this.modal.show(SuccessModalComponent, config);  
        this.bsModalRef.content.redirectOnOk.subscribe(_ => this.redirectToOwnerList());  
      },  
      error: (err: HttpErrorResponse) => {  
        this.errorHandler.handleError(err);  
        this.errorMessage = this.errorHandler.errorMessage;  
      }  
    })  
}
```

Aqui criamos um objeto **OwnerForCreation** que vamos enviar para a **API** com nossa solicitação **POST**. Observe que, neste exemplo, o uso do **pipe** de dados não se restringe apenas a arquivos **HTML**. Claro, para fazê-lo funcionar dentro de um arquivo de componente, precisamos importar o **DatePipe** dentro do arquivo **app.module.ts** e colocá-lo dentro da vetor “**providers**”

Abra o arquivo **src\app\app.module.ts** e faça as alterações abaixo:

```

import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HttpClientModule } from '@angular/common/http';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
import { HomeComponent } from './home/home.component';
import { MenuComponent } from './menu/menu.component';
import { CollapseModule } from 'ngx-bootstrap/collapse';
import { NotFoundComponent } from './error-pages/not-found/not-found.component';
import { OwnerModule } from './owner/owner.module';
import { InternalServerComponent } from './error-pages/internal-server/internal-server.component';
import { DatePipe } from '@angular/common';

@NgModule({
  declarations: [
    AppComponent,
    HomeComponent,
    MenuComponent,
    NotFoundComponent,
    InternalServerComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    BrowserAnimationsModule,
    HttpClientModule,
    CollapseModule.forRoot(),
    OwnerModule
  ],
  providers: [DatePipe],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

Depois de criar o objeto **ownerForCreation**, criamos a **apiUrl**, e chamamos a função **createOwner** do repositório. Agora, uma vez que estamos usando o tipo **OwnerForCreation**, e não o tipo **Owner**, temos que modificar o tipo dentro da função de repositório. Abra o arquivo `\src\app\shared\services\owner-repository.service.ts` e altere a função **createOwner**, conforme abaixo:

```

public createOwner = (route: string, owner: OwnerForCreation) => {
  return this.http.post<Owner>(this.createCompleteRoute(route, this.envUrl.urlAddress),
    owner, this.generateHeaders());
}

```

Dentro da função **subscribe** – com a propriedade **next** – aceitamos a resposta de sucesso da **API**, criamos o objeto de estado inicial para o nosso modal de sucesso e, em seguida, mostramos o modal fornecendo o **SuccessModalComponent** e o objeto **config** como parâmetros. Além disso, usamos o objeto **bsModalRef** para assinar o evento **redirectOnOk** que emitimos do **SucessModalComponent** quando clicamos no botão **OK**.

Redirecionamento

Com essa assinatura, estamos chamando a função **redirectToOwnerList**. Chamamos a mesma função clicando no botão **Cancel** em nosso formulário. Então, vamos adicionar essa função, abaixo da função privada **executeOwnerCreation**:

```
redirectToOwnerList = () => {  
  this.router.navigate(['/owner/list']);  
}
```

Este é o código familiar serve para navegar de volta para o componente anterior. Há outra maneira de fazer isso importando o **Location** do **@angular/common** e injetando-o dentro do construtor e, em seguida, apenas chamando a função **back()** nessa propriedade injetada (**location.back()**). O que você decide usar depende totalmente de você.

Agora basta modificar nosso arquivo **CSS** raiz (styles.css), para mostrar as **tags** de mensagens **** com cor vermelha e o estilo negrito e para envolver as entradas na cor vermelha se elas forem inválidas. Altere o arquivo **owner-create.component.css** conforme código abaixo:

```
em {  
  color: #e71515;  
  font-weight: bold;  
}  
  
.ng-invalid.ng-touched {  
  border-color: red;  
}
```

Testando o funcionamento

Agora, se navegarmos até o componente **CreateOwner**, uma vez que clicarmos dentro de cada elemento de entrada, mas o deixarmos vazio, veremos nossas mensagens de erro:

Account-Owner Home Owner Actions Account Actions

Name of the owner: *Name is required*

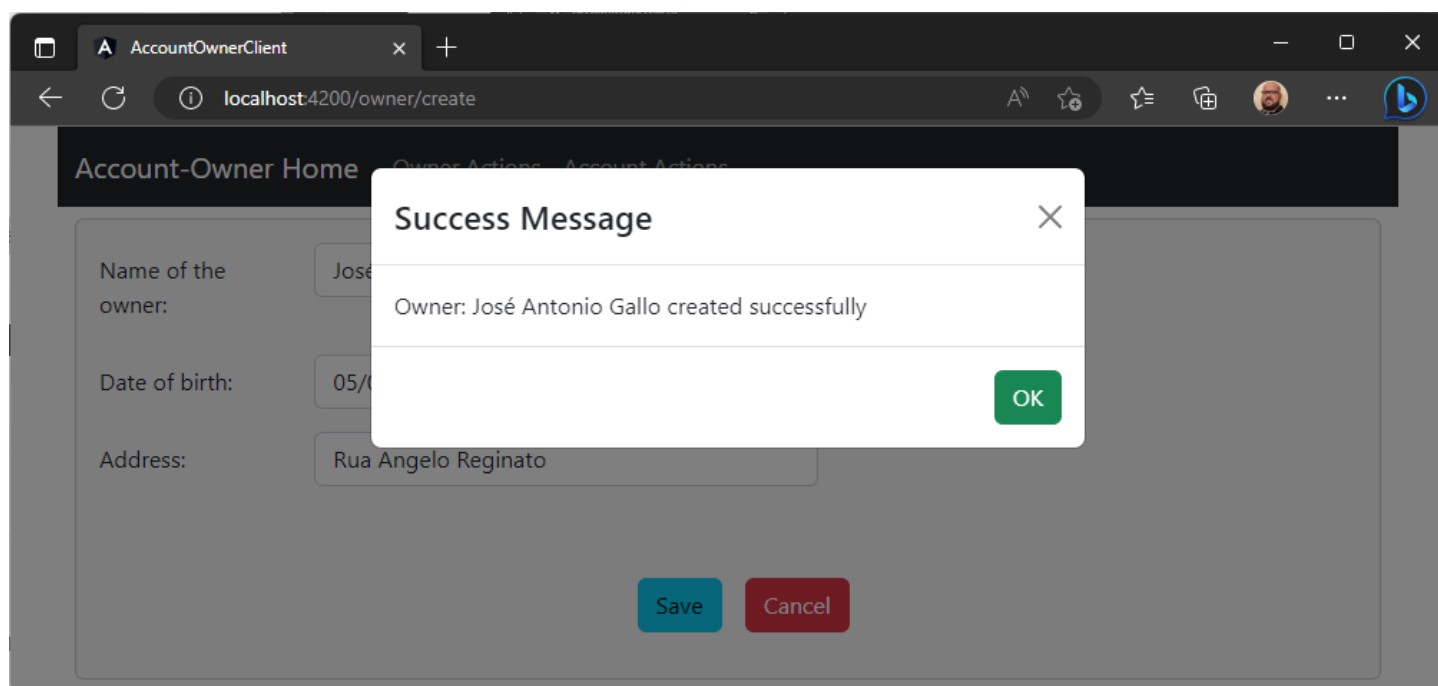
Date of birth: *Date of birth is required*

Address: *Address is required*

Botão desabilitado →

Claro, podemos testar a validação do comprimento máximo também.

Depois de preencher todos os campos e clicarmos no botão **Save**, veremos a mensagem modal de sucesso:



Quando clicarmos no botão **OK**, seremos redirecionados para a página da lista de **owners** e o novo **owner** estará na lista.

Mostrando o componente modal de erro

Abra o arquivo da `\src\app\shared\services\error-handler.service.ts` e faça as modificações, conforme código abaixo:

```
import { HttpResponse } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { Router } from '@angular/router';
import { ErrorModalComponent } from '../modals/error-modal/error-modal.component';
import { BsModalService, ModalOptions } from 'ngx-bootstrap/modal';

@Injectable({
  providedIn: 'root'
})
export class ErrorHandlerService {
  public errorMessage: string = '';

  constructor(private router: Router, private modal: BsModalService) { }

  public handleError = (error: HttpResponse) => {
    if (error.status === 500) {
      this.handle500Error(error);
    }
    else if (error.status === 404) {
      this.handle404Error(error)
    }
  }
}
```

```

    else {
      this.handleOtherError(error);
    }
  }

  private handle500Error = (error: HttpErrorResponse) => {
    this.createErrorMessage(error);
    this.router.navigate(['/500']);
  }

  private handle404Error = (error: HttpErrorResponse) => {
    this.createErrorMessage(error);
    this.router.navigate(['/404']);
  }

  private handleOtherError = (error: HttpErrorResponse) => {
    this.createErrorMessage(error);

    const config: ModalOptions = {
      initialState: {
        modalHeaderText: 'Error Message',
        modalBodyText: this.errorMessage,
        okButtonText: 'OK'
      }
    };
    this.modal.show(ErrorModalComponent, config);
  }

  private createErrorMessage = (error: HttpErrorResponse) => {
    this.errorMessage = error.error ? error.error : error.statusText;
  }
}

```

Agora, se um erro diferente de 500 ou 404 aparecer, vamos mostrar uma mensagem modal para o usuário

Manipulando Solicitações PUT com Angular e ASP.NET Core Web API

Estrutura de pastas e roteamento

Antes de qualquer ação de atualização, precisamos criar nossos arquivos de componentes.

Então, vamos criá-los usando o comando **Angular CLI** que vai criar todos os arquivos e importar o componente **OwnerUpdate** criado no arquivo **owner.module.ts**:

```
ng g component owner/owner-update --skip-tests
```

Agora, para estabelecer a rota para esse componente, precisamos modificar o vetor de rotas no arquivo **owner-routing.module.ts**:

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { OwnerDetailsComponent } from '../owner-details/owner-details.component';

import { OwnerListComponent } from '../owner-list/owner-list.component';
import { OwnerCreateComponent } from '../owner-create/owner-create.component';
import { OwnerUpdateComponent } from '../owner-update/owner-update.component';

const routes: Routes = [
  { path: 'list', component: OwnerListComponent },
  { path: 'details/:id', component: OwnerDetailsComponent },
  { path: 'create', component: OwnerCreateComponent },
  { path: 'update/:id', component: OwnerUpdateComponent }
];

@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class OwnerRoutingModule { }
```

Agora vamos alterar o nosso **owner-list.component.html** e os arquivos **owner-list.component.ts**, para permitir a navegação entre os componentes **OwnerList** e **OwnerUpdate**.

Abra o arquivo **owner-list.component.html** e altere a tag `<td>` que contém o botão Update pelo código abaixo:

```
<td><button type="button" id="update" class="btn btn-success"
  (click)="redirectToUpdatePage(owner.id)">Update</button></td>
```

E no arquivo **owner-list.component.ts** inclua a função **redirectToUpdatePage** abaixo da função **getOwnerDetails**:

```

public redirectToUpdatePage = (id) => {
  const updateUrl: string = `/owner/update/${id}`;
  this.router.navigate([updateUrl]);
}

```

Neste ponto, temos nosso roteamento definido e podemos avançar para lidar com a solicitação **PUT**.

Criando Formulário para Manipular as Solicitações PUT com o Angular

Nosso arquivo **owner-update.component.html** será quase o mesmo que o arquivo **HTML** para criar o proprietário. Já que esse é o caso, vamos começar com a implementação.

Altere o arquivo **owner-update.component.html**, conforme o código abaixo:

```

<div class="container-fluid">
  <form [formGroup]="ownerForm" autocomplete="off" novalidate
  (ngSubmit)="updateOwner(ownerForm.value)">
    <div class="card card-body bg-light mb-2 mt-2">
      <div class="row mb-3">
        <label for="name" class="col-form-label col-md-2">Name of the owner: </label>
        <div class="col-md-5">
          <input type="text" formControlName="name" id="name" class="form-control"
/>

          </div>
          <div class="col-md-5">
            <em *ngIf="validateControl('name')
            && hasError('name', 'required')">Name is required</em>
            <em *ngIf="validateControl('name')
            && hasError('name', 'maxlength')">Maximum allowed length is 60 characters.</em>
            </div>
          </div>

          <div class="mb-3 row">
            <label for="dateOfBirth" class="col-form-label col-md-2">Date of birth:
</label>
            <div class="col-md-5">
              <input type="text" formControlName="dateOfBirth" id="dateOfBirth"
class="form-control" readonly
              bsDatepicker />
            </div>
            <div class="col-md-5">
              <em *ngIf="validateControl('dateOfBirth')
              && hasError('dateOfBirth', 'required')">Date of birth is required</em>
              </div>
            </div>

            <div class="mb-3 row">
              <label for="address" class="col-form-label col-md-2">Address: </label>
              <div class="col-md-5">
                <input type="text" formControlName="address" id="address" class="form-
control" />

```

```

        </div>
        <div class="col-md-5">
            <em *ngIf="validateControl('address')
            && hasError('address', 'required')">Address is required</em>
            <em *ngIf="validateControl('address')
            && hasError('address', 'maxlength')">Maximum allowed length is 100
characters.</em>
        </div>
    </div>

    <br><br>

    <div class="mb-3 row">
        <div class="offset-5 col-md-1">
            <button type="submit" class="btn btn-info"
[disabled]="!ownerForm.valid">Save</button>
        </div>
        <div class="col-md-1">
            <button type="button" class="btn btn-danger"
(click)="redirectToOwnerList()">Cancel</button>
        </div>
    </div>
</div>
</form>
</div>

```

Já sabemos pela criação do componente **createOwner** que o **formGroup** vai conter todos os controles dentro de seu valor. Esse valor é exatamente o que enviamos como parâmetro para a ação **updateOwner**.

Cada elemento de entrada contém um atributo **formControlName** que vamos usar no arquivo de componente para a validação. Além disso, as funções **validateControl** e **hasError** são as funções personalizadas que nos ajudarão a exibir as mensagens de erro (ainda a mesma coisa que fizemos no componente **CreateOwner**).

Agora temos nosso arquivo **HTML** e é hora de implementar a lógica de negócios para o arquivo **owner-update.component**.

Lógica de Negócios Para Manipular Solicitações PUT com Angular

Agora, temos que implementar a lógica para todas as funções chamadas no arquivo de modelo. Vamos adicionar a lógica ao arquivo **owner-update.component.ts**, então vamos tentar dividir a explicação em várias partes. Começemos pelas importações:

```

import { DatePipe } from '@angular/common';
import { HttpResponseResponse } from '@angular/common/http';
import { Component, OnInit } from '@angular/core';
import { FormControl, FormGroup, Validators } from '@angular/forms';
import { ActivatedRoute, Router } from '@angular/router';
import { BsModalRef, BsModalService, ModalOptions } from 'ngx-bootstrap/modal';
import { Owner } from 'src/app/_interfaces/owner.model';
import { OwnerForUpdate } from 'src/app/_interfaces/ownerForUpdate.model';

```

```
import { SuccessModalComponent } from 'src/app/shared/modals/success-modal/success-modal.component';
import { ErrorHandlerService } from 'src/app/shared/services/error-handler.service';
import { OwnerRepositoryService } from 'src/app/shared/services/owner-repository.service';
```

Em seguida, vamos adicionar as propriedades necessárias e injetar os serviços necessários:

```
@Component({
  selector: 'app-owner-update',
  templateUrl: './owner-update.component.html',
  styleUrls: ['./owner-update.component.css']
})
export class OwnerUpdateComponent implements OnInit {
  owner: Owner;
  ownerForm: FormGroup;
  bsModalRef?: BsModalRef;

  constructor(private repository: OwnerRepositoryService, private errorHandler:
ErrorHandlerService,
    private router: Router, private activeRoute: ActivatedRoute, private datePipe: DatePipe,
    private modal: BsModalService) { }
}
```

Para continuar, temos que modificar a função **ngOnInit** e adicionar uma função adicional, abaixo do método construtor:

```
ngOnInit(): void {
  this.ownerForm = new FormGroup({
    name: new FormControl('', [Validators.required, Validators.maxLength(60)]),
    dateOfBirth: new FormControl('', [Validators.required]),
    address: new FormControl('', [Validators.required, Validators.maxLength(100)])
  });

  this.getOwnerById();
}

private getOwnerById = () => {
  const ownerId: string = this.activeRoute.snapshot.params['id'];
  const ownerByIdUri: string = `api/owner/${ownerId}`;

  this.repository.getOwner(ownerByIdUri)
    .subscribe({
      next: (own: Owner) => {
        this.owner = { ...own,
          dateOfBirth: new Date(this.datePipe.transform(own.dateOfBirth, 'MM/dd/yyyy'))
        };
        this.ownerForm.patchValue(this.owner);
      },
      error: (err: HttpErrorResponse) => this.errorHandler.handleError(err)
    })
}
```

Na função **NgOnInit**, instanciamos o **ownerForm** com todos os controles de formulário e adicionamos as regras de validação. Em seguida, chamamos a função **getOwnerByld** para buscar o **owner** com o **id** exato do servidor.

Dentro dessa função, executamos ações familiares. Chamamos com o **id** do **URI** e criamos uma cadeia de caracteres do **URI** da **API**, enviamos a solicitação **GET** e processamos a resposta, seja ela uma resposta bem-sucedida ou de erro.

Uma coisa a prestar atenção é a conversão do valor do campo **dateOfBirth** para o formato que esperamos dentro do nosso controle de entrada. Usamos para transformar o **DatePipe** para aplicar o formato.

Agora, temos que adicionar nossas funções de validação de erros:

```
validateControl = (controlName: string) => {
  if (this.ownerForm.get(controlName).invalid && this.ownerForm.get(controlName).touched)
    return true;

  return false;
}

hasError = (controlName: string, errorName: string) => {
  if (this.ownerForm.get(controlName).hasError(errorName))
    return true;

  return false;
}
```

Essas são as funções familiares para validar os campos de entrada.

Agora, antes de enviarmos a solicitação **PUT**, temos que criar outra interface na pasta **src\app\interfaces**, crie um arquivo com o nome **ownerForUpdate.model.ts** e faça as alterações abaixo:

```
export interface OwnerForUpdate {
  name: string;
  dateOfBirth: string;
  address: string;
}
```

Abra o arquivo **\src\app\shared\services\owner-repository.service.ts** e altere a função **updateOwner**, conforme abaixo:

```
public updateOwner = (route: string, owner: OwnerForUpdate) => {
  return this.http.put(this.createCompleteRoute(route, this.envUrl.urlAddress), owner,
this.generateHeaders());
}
```

Finalmente, vamos retornar ao nosso arquivo de componente **owner-update.component.ts** e codificar a ação de atualização, adicionando abaixo da função **hasError** o seguinte código:

```

public updateOwner = (ownerFormValue) => {
  if (this.ownerForm.valid)
    this.executeOwnerUpdate(ownerFormValue);
}

private executeOwnerUpdate = (ownerFormValue) => {
  const ownerForUpd: OwnerForUpdate = {
    name: ownerFormValue.name,
    dateOfBirth: this.datePipe.transform(ownerFormValue.dateOfBirth, 'yyyy-MM-dd'),
    address: ownerFormValue.address
  }

  const apiUri: string = `api/owner/${this.owner.id}`;

  this.repository.updateOwner(apiUri, ownerForUpd)
    .subscribe({
      next: (_) => {
        const config: ModalOptions = {
          initialState: {
            modalHeaderText: 'Success Message',
            modalBodyText: 'Owner updated successfully',
            okButtonText: 'OK'
          }
        };

        this.bsModalRef = this.modal.show(SuccessModalComponent, config);
        this.bsModalRef.content.redirectOnOk.subscribe(_ => this.redirectToOwnerList());
      },
      error: (err: HttpErrorResponse) => this.errorHandler.handleError(err)
    })
}

public redirectToOwnerList = () => {
  this.router.navigate(['/owner/list']);
}

```

Esta é praticamente a mesma lógica que usamos para a função **createOwner**. Só não temos parâmetros para a propriedade **next**: porque nossa **API** não retorna um objeto como resposta à solicitação **PUT**.

Você pode testar seu projeto e fazer algumas atualizações. Tente criar respostas de sucesso e respostas de erro do servidor para testar os componentes modais também. Depois disso, você pode verificar se a validação do formulário funciona.